

Многопоточное Программирование: Построение атомарных объектов и блокировки

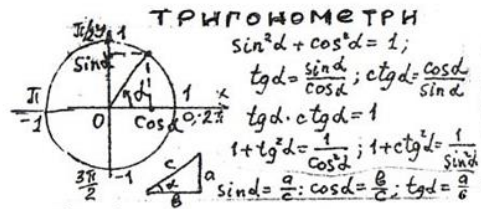
Роман Елизаров, JetBrains, elizarov@gmail.com

Никита Коваль, JetBrains, ndkoval@ya.ru

ИТМО 2020



ITMO UNIVERSITY



$\sin(\alpha \pm \beta) = \sin \alpha \cos \beta \pm \cos \alpha \sin \beta$
 $\cos(\alpha \pm \beta) = \cos \alpha \cos \beta \mp \sin \alpha \sin \beta$
 $\operatorname{tg}(\alpha \pm \beta) = \frac{\operatorname{tg} \alpha \pm \operatorname{tg} \beta}{1 \mp \operatorname{tg} \alpha \operatorname{tg} \beta}; \operatorname{ctg}(\alpha \pm \beta) = \frac{\operatorname{ctg} \alpha \operatorname{ctg} \beta \mp 1}{\operatorname{ctg} \beta \pm \operatorname{ctg} \alpha}$

$\sin \frac{\alpha}{2} = \pm \sqrt{\frac{1 - \cos \alpha}{2}}; \cos \frac{\alpha}{2} = \pm \sqrt{\frac{1 + \cos \alpha}{2}}$
 $\operatorname{tg} \frac{\alpha}{2} = \pm \sqrt{\frac{1 - \cos \alpha}{1 + \cos \alpha}}; \operatorname{ctg} \frac{\alpha}{2} = \pm \sqrt{\frac{1 + \cos \alpha}{1 - \cos \alpha}}$

$\sin 2\alpha = 2 \sin \alpha \cos \alpha$
 $\cos 2\alpha = \cos^2 \alpha - \sin^2 \alpha = 1 - 2 \sin^2 \alpha = 2 \cos^2 \alpha - 1$
 $\operatorname{tg} 2\alpha = \frac{2 \operatorname{tg} \alpha}{1 - \operatorname{tg}^2 \alpha}; \operatorname{ctg} 2\alpha = \frac{\operatorname{ctg} \alpha - \operatorname{ctg} \alpha}{2 \operatorname{ctg} \alpha}$

$\sin \alpha + \sin \beta = 2 \sin \frac{\alpha + \beta}{2} \cos \frac{\alpha - \beta}{2}$
 $\sin \alpha - \sin \beta = 2 \cos \frac{\alpha + \beta}{2} \sin \frac{\alpha - \beta}{2}$
 $\cos \alpha + \cos \beta = 2 \cos \frac{\alpha + \beta}{2} \cos \frac{\alpha - \beta}{2}$
 $\cos \alpha - \cos \beta = -2 \sin \frac{\alpha + \beta}{2} \sin \frac{\alpha - \beta}{2}$

$\sin \alpha \cdot \sin \beta = \frac{1}{2} [\cos(\alpha - \beta) - \cos(\alpha + \beta)]$
 $\cos \alpha \cdot \cos \beta = \frac{1}{2} [\cos(\alpha - \beta) + \cos(\alpha + \beta)]$
 $\sin \alpha \cdot \cos \beta = \frac{1}{2} [\sin(\alpha - \beta) + \sin(\alpha + \beta)]$

α	0	30° π/6	45° π/4	60° π/3	90° π/2	120° 2π/3	135° 3π/4	150° 5π/6	180° π
sin α	0	1/2	√2/2	√3/2	1	√3/2	√2/2	1/2	0
cos α	1	√3/2	√2/2	1/2	0	-1/2	-√2/2	-√3/2	-1
tg α	0	1/√3	1	√3	-	-√3	-1	-1/√3	0
ctg α	-	√3	1	1/√3	0	-1/√3	-1	-√3	-

$\sin(\frac{\pi}{2} \pm \alpha) = \cos \alpha$
 $\sin(\frac{3\pi}{2} \pm \alpha) = -\cos \alpha$
 $\sin(\pi \pm \alpha) = -\sin \alpha$
 $\sin(\frac{\pi}{2} - \alpha) = \cos \alpha$
 $\sin(\frac{3\pi}{2} - \alpha) = -\cos \alpha$
 $\sin(\pi - \alpha) = \sin \alpha$

$-\frac{\pi}{2} \leq \arcsin a \leq \frac{\pi}{2}; |a| \leq 1$
 $0 \leq \arccos a \leq \pi; |a| \leq 1$
 $-\frac{\pi}{2} \leq \operatorname{arctg} a < \frac{\pi}{2}; a \text{ — любое}$
 $0 < \operatorname{arctg} a < \pi$

$\sin(-x) = -\sin x; \arcsin(-a) = -\arcsin a$
 $\cos(-x) = \cos x; \arccos(-a) = \pi - \arccos a$
 $\operatorname{tg}(-x) = -\operatorname{tg} x; \operatorname{arctg}(-a) = -\operatorname{arctg} a$
 $\operatorname{ctg}(-x) = -\operatorname{ctg} x; \operatorname{arctg}(-a) = \pi - \operatorname{arctg} a$



АЛГЕБРА

$a^2 - b^2 = (a - b)(a + b)$
 $(a + b)^2 = a^2 + 2ab + b^2$
 $(a - b)^2 = a^2 - 2ab + b^2$
 $(a + b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$
 $(a - b)^3 = a^3 - 3a^2b + 3ab^2 - b^3$
 $a^2 + b^2 = (a + b)(a - b) + 2ab$
 $a^3 - b^3 = (a - b)(a^2 + ab + b^2)$

Степени:

$2^n \cdot a^m = a^{n+m}; (\frac{a}{b})^n = \frac{a^n}{b^n}; a^{-n} = \frac{1}{a^n}$
 $\frac{a^m}{a^n} = a^{m-n}; (\frac{a}{b})^n = \frac{a^n}{b^n}; a^{-n} = \frac{1}{a^n}$
 $a^1 = a; a^0 = 1$

Корни:

$\sqrt[n]{a} = a^{\frac{1}{n}}; \sqrt[n]{a^m} = a^{\frac{m}{n}}$
 $\sqrt[n]{a} \cdot \sqrt[n]{b} = \sqrt[n]{ab}; \frac{\sqrt[n]{a}}{\sqrt[n]{b}} = \sqrt[n]{\frac{a}{b}}; \sqrt[n]{a^k} = \sqrt[n]{a}^k$
 $\sqrt[n]{\sqrt[n]{a}} = \sqrt[n^2]{a}; (\sqrt[n]{a})^m = \sqrt[n]{a^m}; \sqrt[n]{a^m} = a^{\frac{m}{n}}$

Логарифмы:

$\log_a b = x \Leftrightarrow a^x = b; a \neq 1$
 $\log_a a = 1; \log_a 1 = 0$
 $\log_a a^n = n; \log_a b^n = n \log_a b$
 $\log_a x = \log x + \log a; \log_a \frac{x}{y} = \log_a x - \log_a y$

$\log_a xy = \log_a x + \log_a y$
 $\log_a \frac{x}{y} = \log_a x - \log_a y$
 $\log_a \sqrt[n]{x} = \frac{1}{n} \log_a x$
 $\log_a b = \frac{\log x}{\log a}; \log_a b = \ln b; e \approx 2,718...$

Прогрессии:

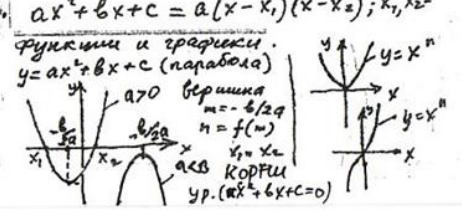
Арифметическая:
 $a_n = a_1 + (n-1)d$
 $S_n = \frac{(a_1 + a_n) \cdot n}{2}$
Геометрическая:
 $a_n = a_1 \cdot q^{n-1}$
 $S_n = \frac{a_1(1 - q^n)}{1 - q}; q \neq 1$

КВУР: $ax^2 + bx + c = 0; D = b^2 - 4ac$

$D \geq 0 \Rightarrow x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}; D < 0, \text{ нет решений}$
 $x_1 \cdot x_2 = \frac{c}{a}; x_1 + x_2 = -\frac{b}{a}$

Модуль:

$|a| = \begin{cases} a, & \text{если } a \geq 0 \\ -a, & \text{если } a < 0 \end{cases}$
 $|a|^2 = a^2; |\frac{a}{b}| = \frac{|a|}{|b|}$
 $|a| \leq b \Leftrightarrow -b \leq a \leq b; |a| \geq b \Leftrightarrow a \geq b \text{ или } a \leq -b$
 $\sqrt{a^2} = |a|$
 $ax^2 + bx + c = a(x - x_1)(x - x_2); x_1, x_2$



Производная

$y' = f'(x) = \lim_{\Delta x \rightarrow 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$
 касат. к графику функции в т. $x = x_0$
 $y = f(x); y' = f'(x_0)(x - x_0) + f(x_0)$
 $f'(x) = \operatorname{tg} \alpha = k$ — углов. коэф.

Правила дифференцирования:

$(C \cdot f(x))' = C \cdot f'(x); (f(g(x)))' = f'(g(x)) \cdot g'(x)$
 $(u \pm v)' = u' \pm v'; (f(kx + b))' = k \cdot f'(kx + b)$
 $(uv)' = u'v + uv'; (\frac{u}{v})' = \frac{u'v - uv'}{v^2}$

Таблица производных:

$(C)' = 0; (x)' = 1; (kx)' = k$
 $(x^a)' = a \cdot x^{a-1}; (\frac{1}{x})' = -\frac{1}{x^2}; (\sqrt{x})' = \frac{1}{2\sqrt{x}}$
 $(e^x)' = e^x; (a^x)' = a^x \cdot \ln a$
 $(\ln x)' = \frac{1}{x}; (\log_a x)' = \frac{1}{x \ln a}$

$(\sin x)' = \cos x; (\cos x)' = -\sin x$
 $(\operatorname{tg} x)' = \frac{1}{\cos^2 x}; (\operatorname{ctg} x)' = -\frac{1}{\sin^2 x}$
 $(\arcsin x)' = \frac{1}{\sqrt{1-x^2}}; (\arccos x)' = -\frac{1}{\sqrt{1-x^2}}$
 $(\operatorname{arctg} x)' = \frac{1}{1+x^2}; (\operatorname{arctg} x)' = \frac{1}{1+x^2}$

В физике:

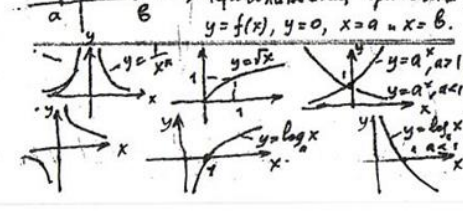
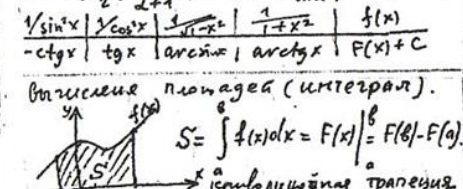
$v(t) = s'(t); a(t) = v'(t) = s''(t)$
 $i(t) = q'(t); e_i = -\varphi'(t)$

Таблица первообразных:

$\int k dx = kx; \int \frac{1}{x} dx = \ln|x|; \int e^x dx = e^x$
 $\int \sin x dx = -\cos x; \int \cos x dx = \sin x$
 $\int \frac{1}{\sin^2 x} dx = -\operatorname{ctg} x; \int \frac{1}{\cos^2 x} dx = \operatorname{tg} x$
 $\int \frac{1}{\sqrt{1-x^2}} dx = \arcsin x; \int \frac{1}{1+x^2} dx = \operatorname{arctg} x$
 $\int f(x) \cdot g(x) dx = F(x) \cdot g(x) - \int F'(x) \cdot g(x) dx$

Интегралы:

$S = \int_a^b f(x) dx = F(b) - F(a)$
 $y = f(x); y = 0; x = a; x = b$
 $y = \frac{1}{x}; y = \ln x; y = a^x; y = \log_a x$



Тригонометрические уравнения и неравенства

$\sin x = a, |a| \leq 1; x = (-1)^n \arcsin a + \pi k$
 $\cos x = a, |a| \leq 1; x = \pm \arccos a + 2\pi k$
 $\operatorname{tg} x = a; x = \operatorname{arctg} a + \pi k$
 $\sin x = 0; x = \pi n$
 $\sin x = -1; x = -\frac{\pi}{2} + 2\pi n$
 $\sin x = 1; x = \frac{\pi}{2} + 2\pi n$
 $\cos x = 0; x = \frac{\pi}{2} + \pi n$
 $\cos x = 1; x = 2\pi n$
 $\cos x = -1; x = \pi + 2\pi n$
 $n \in \mathbb{Z}$

Логарифмические уравнения и неравенства:

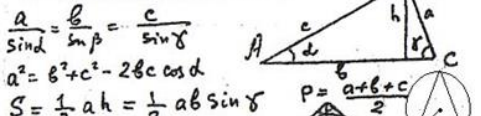
$\log_a x = b \Rightarrow x = a^b; a > 0, a \neq 1$
 $a^x = b \Rightarrow x = \log_a b$
 $a^{f(x)} = a^{g(x)} \Leftrightarrow f(x) = g(x)$
 $\log_a f(x) = \log_a g(x) \Leftrightarrow f(x) = g(x); f(x) > 0, g(x) > 0$

$\log_a f(x) < \log_a g(x) \Leftrightarrow \begin{cases} f(x) > g(x), & \text{если } a > 1 \\ f(x) < g(x), & \text{если } a < 1 \end{cases}$

$\log_a f(x) < \log_a g(x) \Leftrightarrow \begin{cases} f(x) > g(x), & \text{если } a > 1 \\ f(x) < g(x), & \text{если } a < 1 \end{cases}$

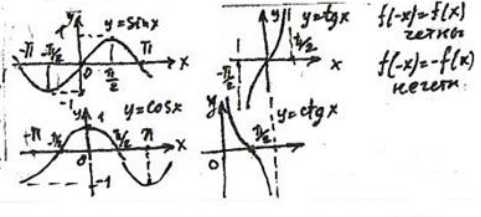
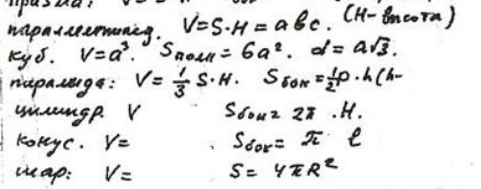
Геометрия

$\frac{a}{\sin \alpha} = \frac{b}{\sin \beta} = \frac{c}{\sin \gamma}$
 $a^2 = b^2 + c^2 - 2bc \cos \alpha$
 $S = \frac{1}{2} ah = \frac{1}{2} ab \sin \gamma$
 $S = \sqrt{p(p-a)(p-b)(p-c)}$
 $S = p \cdot r$ (r — радиус вписан. окружности)
 $R = \frac{abc}{4S}$ (R — радиус описан. окружности)



Стереометрия

пирамида: $V = \frac{1}{3} S_{\text{осн}} \cdot H$
 параллелепипед: $V = S_{\text{осн}} \cdot H = abc$
 куб: $V = a^3; S_{\text{полн}} = 6a^2; d = a\sqrt{3}$
 параллелепипед: $V = \frac{1}{3} S_{\text{осн}} \cdot H$
 цилиндр: $V = S_{\text{осн}} \cdot H$
 конус: $V = \frac{1}{3} S_{\text{осн}} \cdot H$
 шар: $V = \frac{4}{3} \pi R^3$



Наведем порядок в формализме

Сложные и составные операции

Формализация сложных операций

- **Вспомним: исполнение** системы – это пара (H, \rightarrow_H)
 - иррефлексивное $\forall e \in H : e \nrightarrow_H e$
 - антисимметричное $\forall e, f \in H : e \rightarrow_H f \Rightarrow f \nrightarrow_H e$
 - транзитивное $\forall e, f, g \in H :$
 $e \rightarrow_H f \ \& \ f \rightarrow_H g \Rightarrow e \rightarrow_H g$
- **Операция** (сложная) состоит из двух **событий** (простых):
 - $inv(e)$ – вызов операции
 - $res(e)$ – ответ на операцию (результат)
 - Все события **полностью** упорядочены отношением $<_H$

Формализация сложных операций

- Вспомним: исполнение системы – это пара (H, \rightarrow_H)
 - иррефлексивное $\forall e \in H : e \nrightarrow_H e$
 - антисимметричное $\forall e, f \in H : e \rightarrow_H f \Rightarrow f \nrightarrow_H e$
 - транзитивное $\forall e, f, g \in H :$
 $e \rightarrow_H f \ \& \ f \rightarrow_H g \Rightarrow e \rightarrow_H g$
- **Операция** $e \in H$ может быть *сложной*
 - Даже чтение переменной из памяти это продолжительное по времени действие, много шагов

Как это моделировать в нашем формализме, чтобы доказывать факты о сложных операциях?

Формализация сложных операций

- Далее будет рассматривать расширенное множество событий
 - **Событие** — неделимое, простое *физическое* действие
 - Множество событий обозначим G
 - **Каждая операция это множество событий $e \subset G$**
- Из всех **событий** в **операции** e выделим два наиболее важных
 - **Вызов операции** $\text{inv}(e) \in G$
 - **Завершение операции** $\text{res}(e) \in G$

Формализация сложных операций

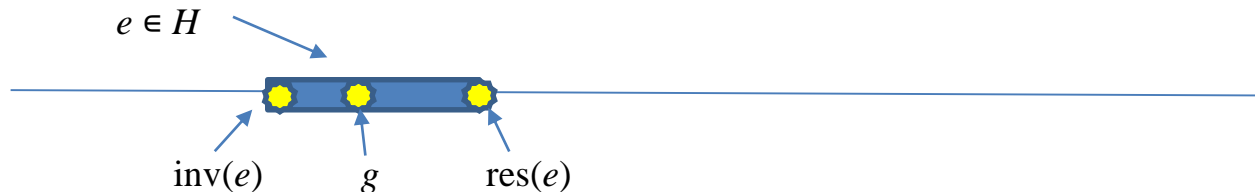
- Определим **декомпозицию исполнения** как пятерку

$$(H, G, \rightarrow_G, \text{inv}, \text{res})$$

- H это множество **операций** ($\forall e \in H : e \subset G$)
- G это множество **событий**
- \rightarrow_G отношение «произошло до» на **событиях** из G
 - антирефлексивное, асимметричное, транзитивное
- inv, res это функции из H в G такие что:

$$\forall e \in H : \text{inv}(e) \rightarrow_G \text{res}(e)$$

$$\forall e \in H, g \in e, g \neq \text{inv}(e), g \neq \text{res}(e) : \text{inv}(e) \rightarrow_G g \rightarrow_G \text{res}(e)$$



Все $g \in e$ не обязательно должны быть упорядочены,
но все $\text{inv}(e)$ и $\text{res}(e)$ упорядочены

Формализация сложных операций

- Определим **декомпозицию исполнения** как пятерку

$$(H, G, \rightarrow_G, \text{inv}, \text{res})$$

- H это множество **операций** ($\forall e \in H : e \subset G$)
- G это множество **событий**
- \rightarrow_G отношение «произошло до» на **событиях** из G
 - антирефлексивное, асимметричное, транзитивное
- inv, res это функции из H в G такие что:

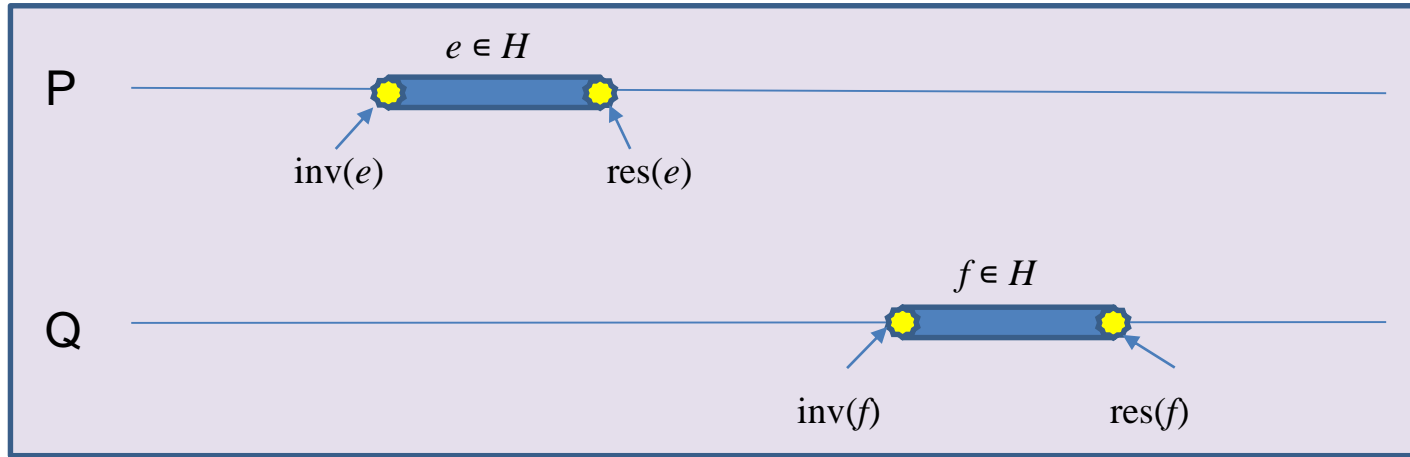
$$\forall e \in H : \text{inv}(e) \rightarrow_G \text{res}(e)$$

$$\forall e \in H, g \in e, g \neq \text{inv}(e), g \neq \text{res}(e) : \text{inv}(e) \rightarrow_G g \rightarrow_G \text{res}(e)$$

Определяем «произошло до» на операциях

$$\forall e, f \in H : e \rightarrow_H f \stackrel{\text{def}}{=} \text{res}(e) \rightarrow_G \text{inv}(f)$$

Формализация сложных операций

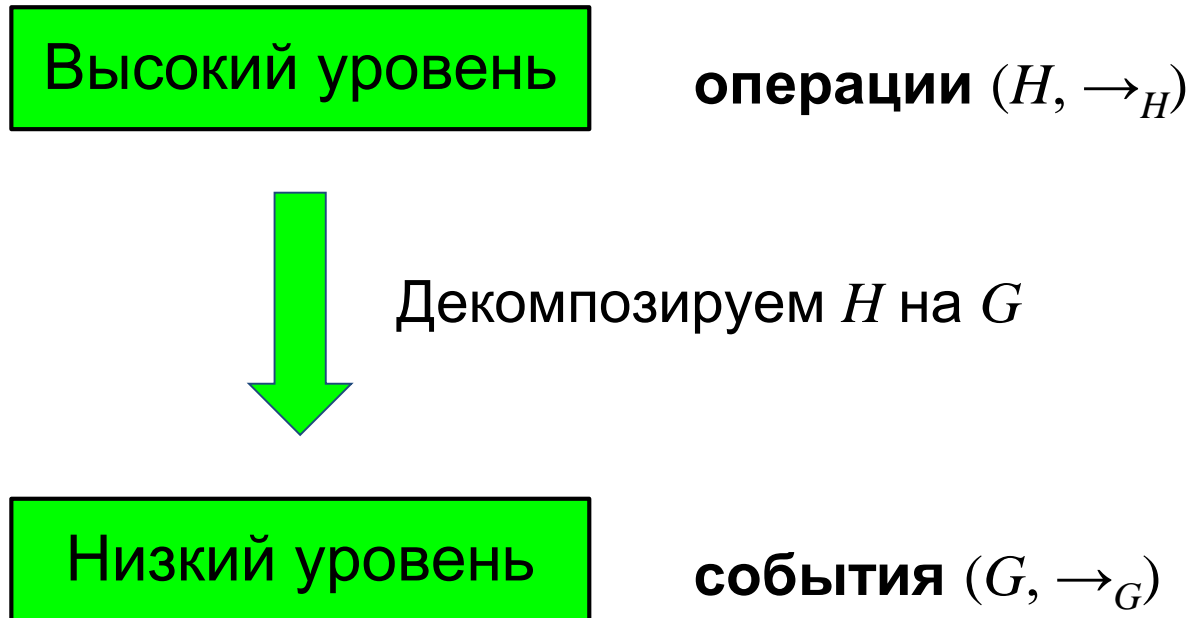


$$inv(e) \rightarrow_G \mathbf{res(e)} \rightarrow_G inv(f) \rightarrow_G res(f)$$

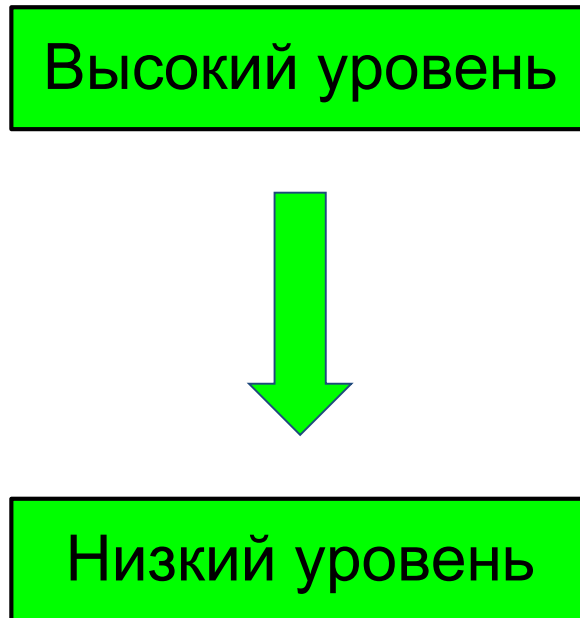


$$e \rightarrow_H f$$

Декомпозиция наглядно



Декомпозиция наглядно

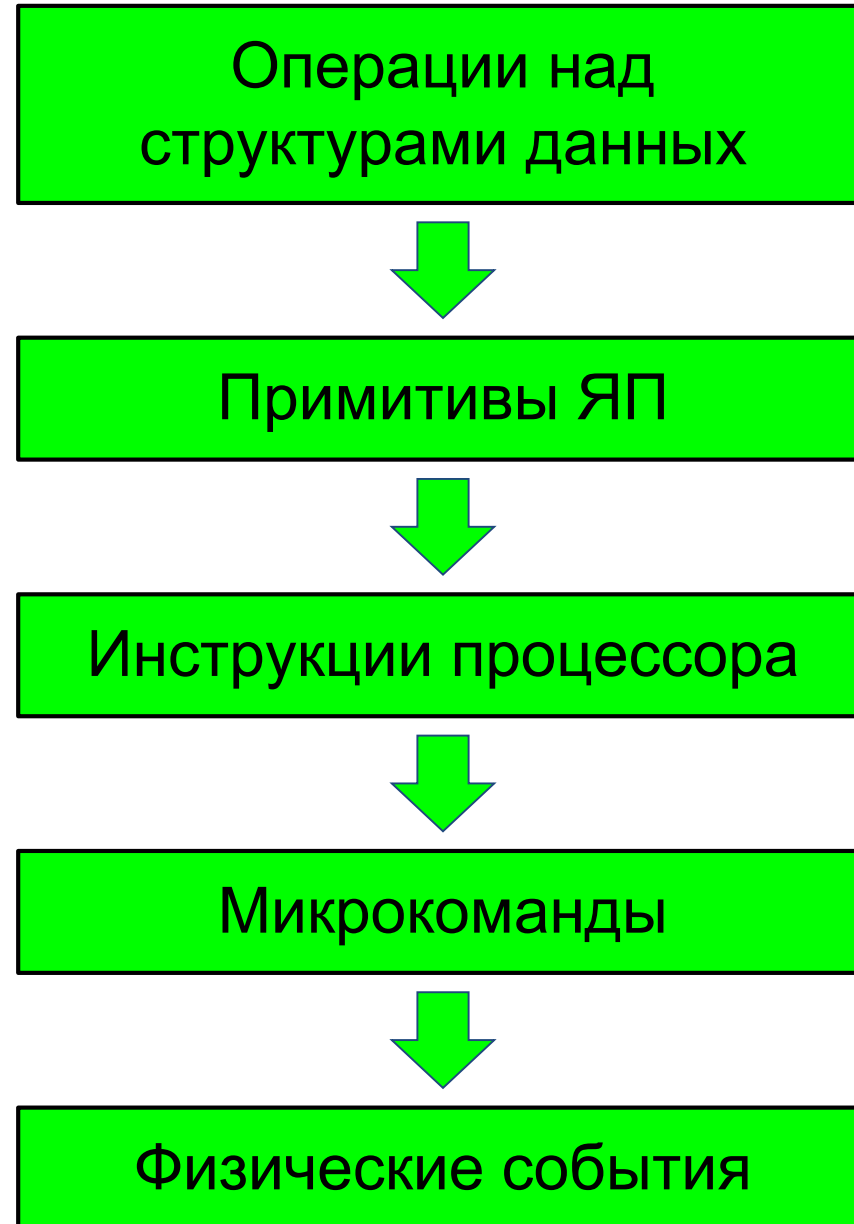


операции (H, \rightarrow_H)

Добавляем H, inv, res

события (G, \rightarrow_G)

На практике



Формализация линеаризации

- Исполнение (H, \rightarrow_H) **линеаризуемо**, если можно найти исполнение $(L(H), \rightarrow_{L(H)})$, называемое **линеаризацией** H , такое что
 - $L(H)$ эквивалентно H : $L(H) = H$ (состоит из тех же операций)
 - $L(H)$ сохраняет отношение “**произошло до**” из H :
$$\forall e, f \in H : e \rightarrow_H f \Rightarrow e \rightarrow_{L(H)} f$$
 - $L(H)$ это **последовательное** исполнение:
$$\forall e, f \in H : e = f \vee e \rightarrow_{L(H)} f \vee f \rightarrow_{L(H)} e$$
 - $L(H)$ это **допустимое** исполнение, то есть выполнены **последовательные спецификации** всех объектов

Точки линеаризации сложных операций

- **Дано:**

- Декомпозиция $(H, G, \rightarrow_G, \text{inv}, \text{res})$

- **Точки линеаризации** это функция p из H в G

$$p(e) \in G$$

каждой **операции** сопоставляется одно **событие**

- Заданное вместе с **линеаризаций** $(L(P), \rightarrow_{L(P)})$
- где P это множество всех **точек линеаризации**

$$P = p(H) \subset G$$

- **Значит:**

- $\rightarrow_{L(P)}$ полный порядок над всеми точками линеаризации P
- $\rightarrow_{L(P)}$ сохраняет порядок на **событиях** \rightarrow_G

Теорема о точках линеаризации

- **Дано:**

- Декомпозиция $(H, G, \rightarrow_G, \text{inv}, \text{res})$

- **Доказать:**

- Исполнение H линеаризуемо $(L(H), \rightarrow_{L(H)})$ **тогда и только тогда*** нет (с оговорками)

- Можно выбрать **точки линеаризации** $p(e) \in G$
- Согласовано с линеаризацией всех **операций**

$$\forall e, f \in H: e \rightarrow_{L(H)} f \Leftrightarrow p(e) \rightarrow_{L(P)} p(f)$$

- **Другими словами:**

- Можно найти точки линеаризации и полный порядок над ними, который согласован с частичным порядком над событиями и согласован с линеаризацией соответствующих операций

Есть точки линеаризации \Rightarrow линеаризуемо

- **Даны:**

- Точки линеаризации $p(e) \in G$ для всех $e \in H$
- Полный порядок над ними $\rightarrow_{L(P)}$ удовлетворяющий последовательные спецификации всех объектов

- **Найти:**

- Линеаризацию операций $(L(H), \rightarrow_{L(H)})$

Есть точки линеаризации \Rightarrow линеаризуемо

- **Даны:**

- Точки линеаризации $p(e) \in G$ для всех $e \in H$
- Полный порядок над ними $\rightarrow_{L(P)}$ удовлетворяющий последовательные спецификации всех объектов

- **Найти:**

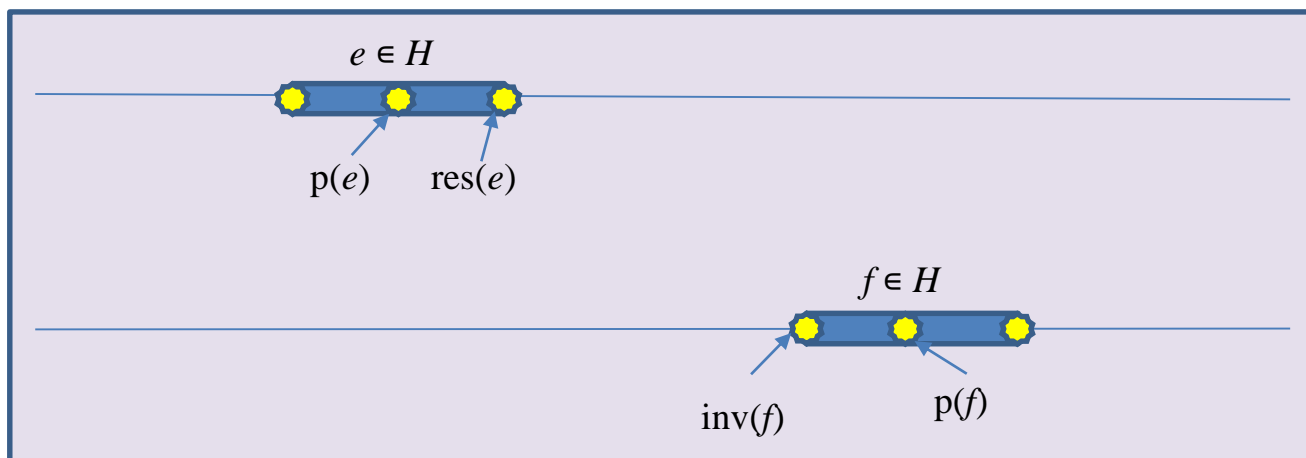
- Линеаризацию операций $(L(H), \rightarrow_{L(H)})$

- **Доказательство:**

- Определим $L(H)$ через $L(P)$ согласованным образом
$$\forall e, f \in H : e \rightarrow_{L(H)} f \Leftrightarrow p(e) \rightarrow_{L(P)} p(f)$$
- ✓ **Последовательное исполнение** (полный порядок)
- ✓ **Допустимое исполнение** (последовательные спецификации)
- Осталось доказать что $\rightarrow_{L(H)}$ **сохраняет** \rightarrow_H

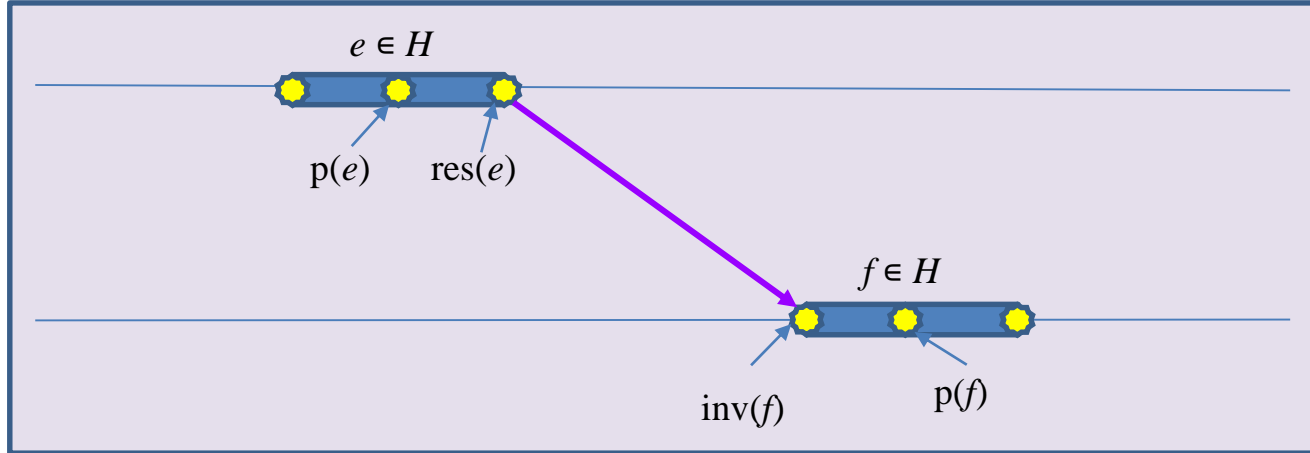
Есть точки линеаризации \Rightarrow линеаризуемо

- Доказать что $\rightarrow_{L(H)}$ **сохраняет** \rightarrow_H
- Возьмем: $\forall e, f \in H : e \rightarrow_H f$
- Покажем: $e \rightarrow_{L(H)} f$



Есть точки линеаризации \Rightarrow линеаризуемо

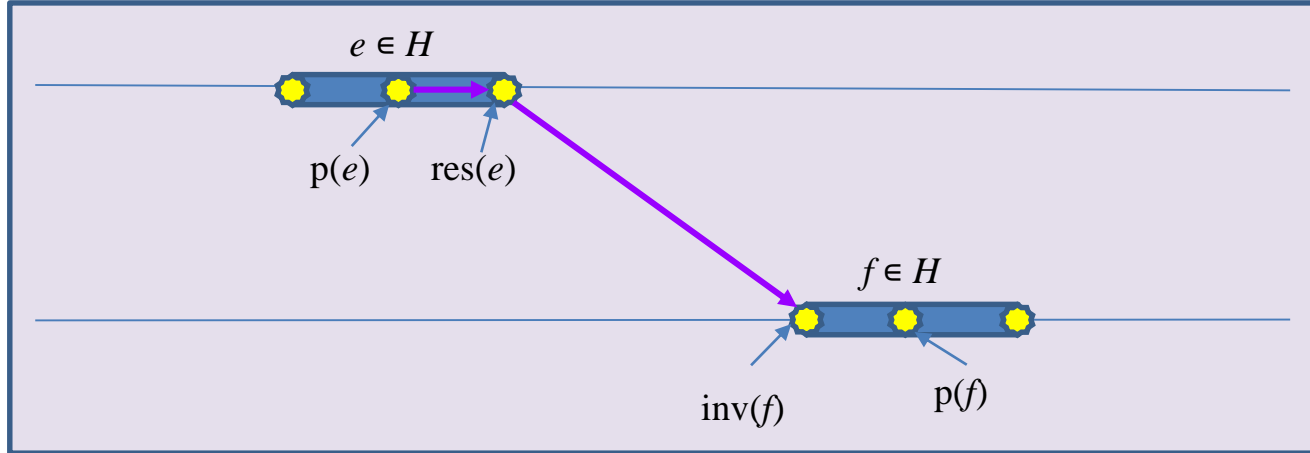
- Доказать что $\rightarrow_{L(H)}$ **сохраняет** \rightarrow_H
- Возьмем: $\forall e, f \in H : e \rightarrow_H f$
- Покажем: $e \rightarrow_{L(H)} f$



1. $res(e) \rightarrow_G inv(f)$ по определению $e \rightarrow_H f$

Есть точки линеаризации \Rightarrow линеаризуемо

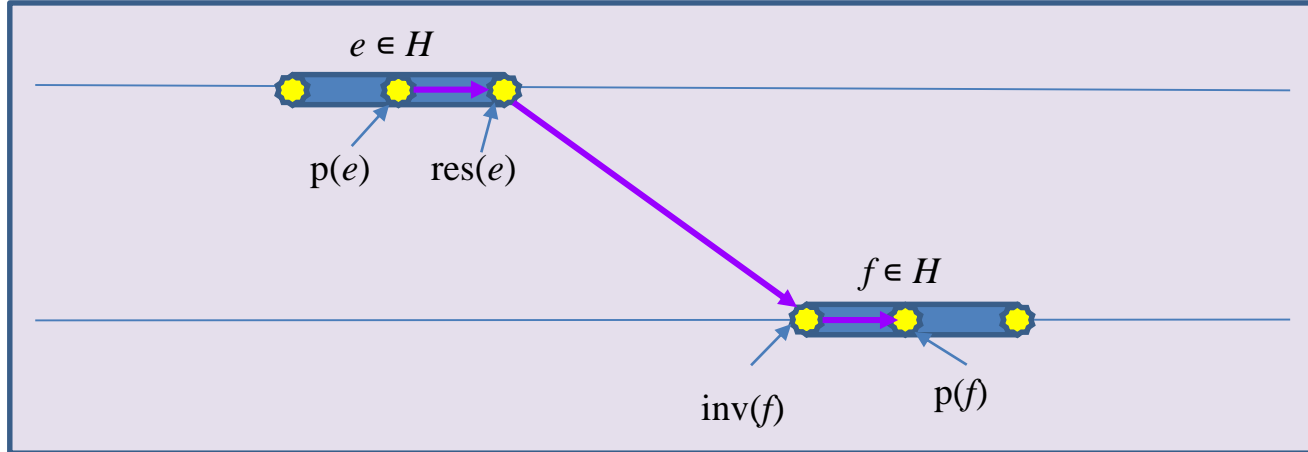
- Доказать что $\rightarrow_{L(H)}$ **сохраняет** \rightarrow_H
- Возьмем: $\forall e, f \in H : e \rightarrow_H f$
- Покажем: $e \rightarrow_{L(H)} f$



1. $res(e) \rightarrow_G inv(f)$ по определению $e \rightarrow_H f$
2. $p(e) \rightarrow_G res(e)$ по определению сложных операций

Есть точки линеаризации \Rightarrow линеаризуемо

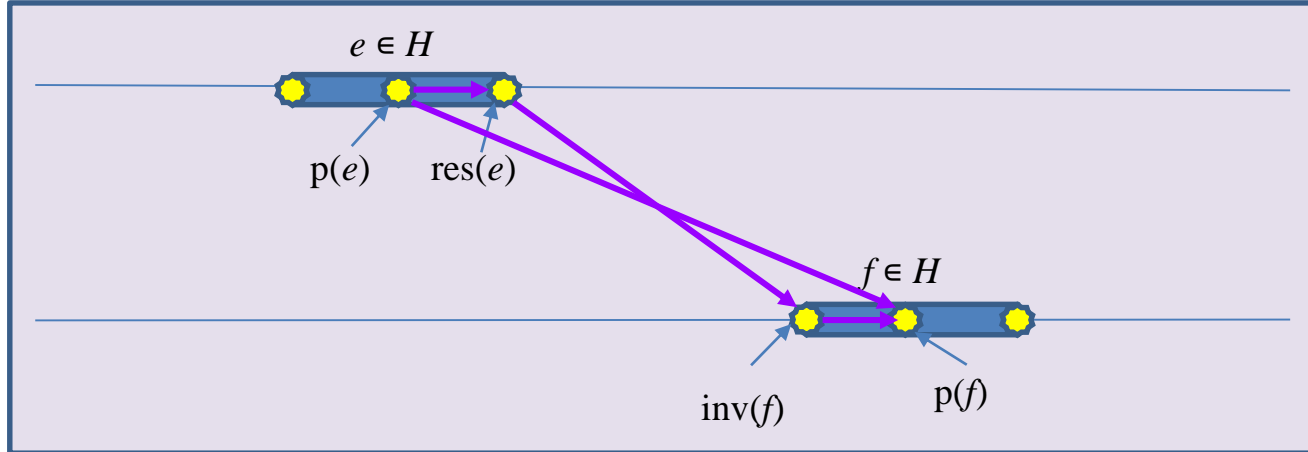
- Доказать что $\rightarrow_{L(H)}$ **сохраняет** \rightarrow_H
- Возьмем: $\forall e, f \in H : e \rightarrow_H f$
- Покажем: $e \rightarrow_{L(H)} f$



1. $\text{res}(e) \rightarrow_G \text{inv}(f)$ по определению $e \rightarrow_H f$
2. $p(e) \rightarrow_G \text{res}(e)$ по определению сложных операций
3. $\text{inv}(f) \rightarrow_G p(f)$ по определению сложных операций

Есть точки линеаризации \Rightarrow линеаризуемо

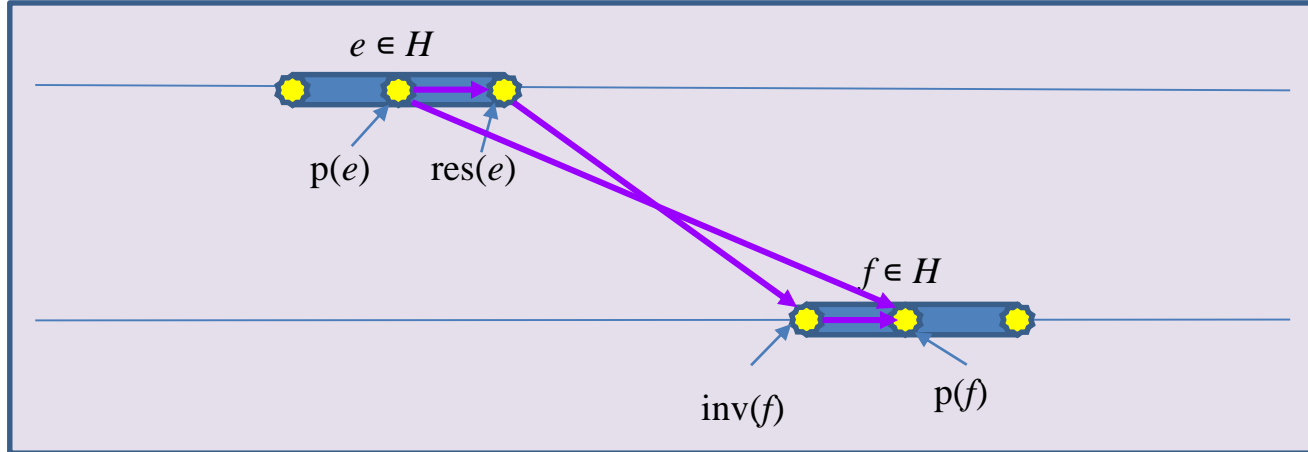
- Доказать что $\rightarrow_{L(H)}$ **сохраняет** \rightarrow_H
- Возьмем: $\forall e, f \in H : e \rightarrow_H f$
- Покажем: $e \rightarrow_{L(H)} f$



- | | |
|----------------------------------|------------------------------------|
| 1. $res(e) \rightarrow_G inv(f)$ | по определению $e \rightarrow_H f$ |
| 2. $p(e) \rightarrow_G res(e)$ | по определению сложных операций |
| 3. $inv(f) \rightarrow_G p(f)$ | по определению сложных операций |
| 4. $p(e) \rightarrow_G p(f)$ | из-за транзитивности |

Есть точки линеаризации \Rightarrow линеаризуемо

- Доказать что $\rightarrow_{L(H)}$ **сохраняет** \rightarrow_H
- Возьмем: $\forall e, f \in H : e \rightarrow_H f$
- Покажем: $e \rightarrow_{L(H)} f$



1. $res(e) \rightarrow_G inv(f)$
2. $p(e) \rightarrow_G res(e)$
3. $inv(f) \rightarrow_G p(f)$
4. $p(e) \rightarrow_G p(f)$
5. $p(e) \rightarrow_{L(P)} p(f)$

по определению $e \rightarrow_H f$

по определению сложных операций

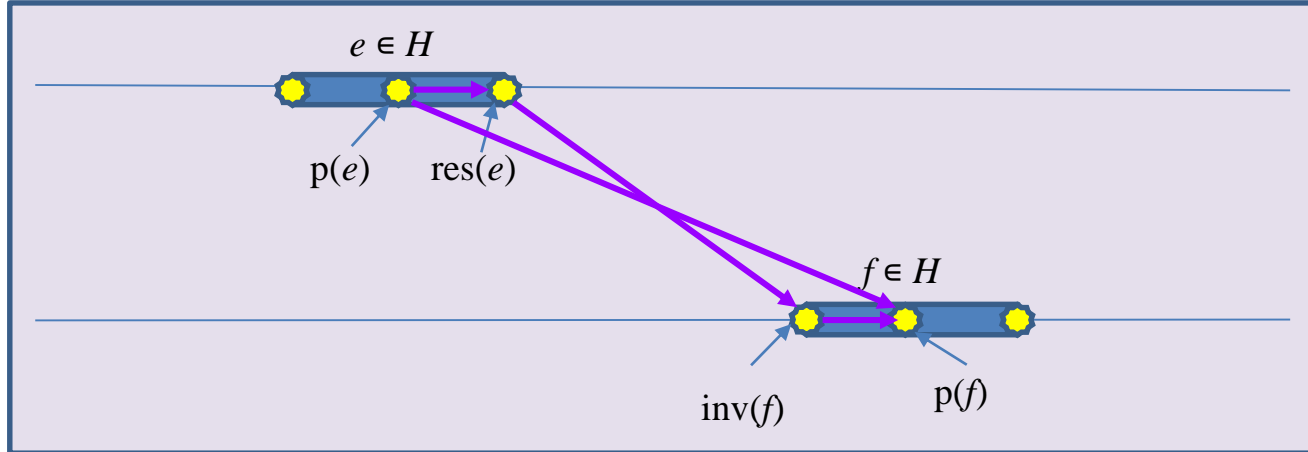
по определению сложных операций

из-за транзитивности

так как линеаризация $L(P)$ сохраняет G

Есть точки линеаризации \Rightarrow линеаризуемо

- Доказать что $\rightarrow_{L(H)}$ **сохраняет** \rightarrow_H
- Возьмем: $\forall e, f \in H : e \rightarrow_H f$
- Покажем: $e \rightarrow_{L(H)} f$



1. $res(e) \rightarrow_G inv(f)$
2. $p(e) \rightarrow_G res(e)$
3. $inv(f) \rightarrow_G p(f)$
4. $p(e) \rightarrow_G p(f)$
5. $p(e) \rightarrow_{L(P)} p(f)$
6. $e \rightarrow_{L(H)} f$

по определению $e \rightarrow_H f$

по определению сложных операций

по определению сложных операций

из-за транзитивности

так как линеаризация $L(P)$ сохраняет G

по определению $L(H)$

Линеаризуемо \Rightarrow есть точки линеаризации

- **Даны:**
 - Линеаризация операций $(L(H), \rightarrow_{L(H)})$
- **Найти:**
 - Точки линеаризации $p(e) \in G$ для всех $e \in H$

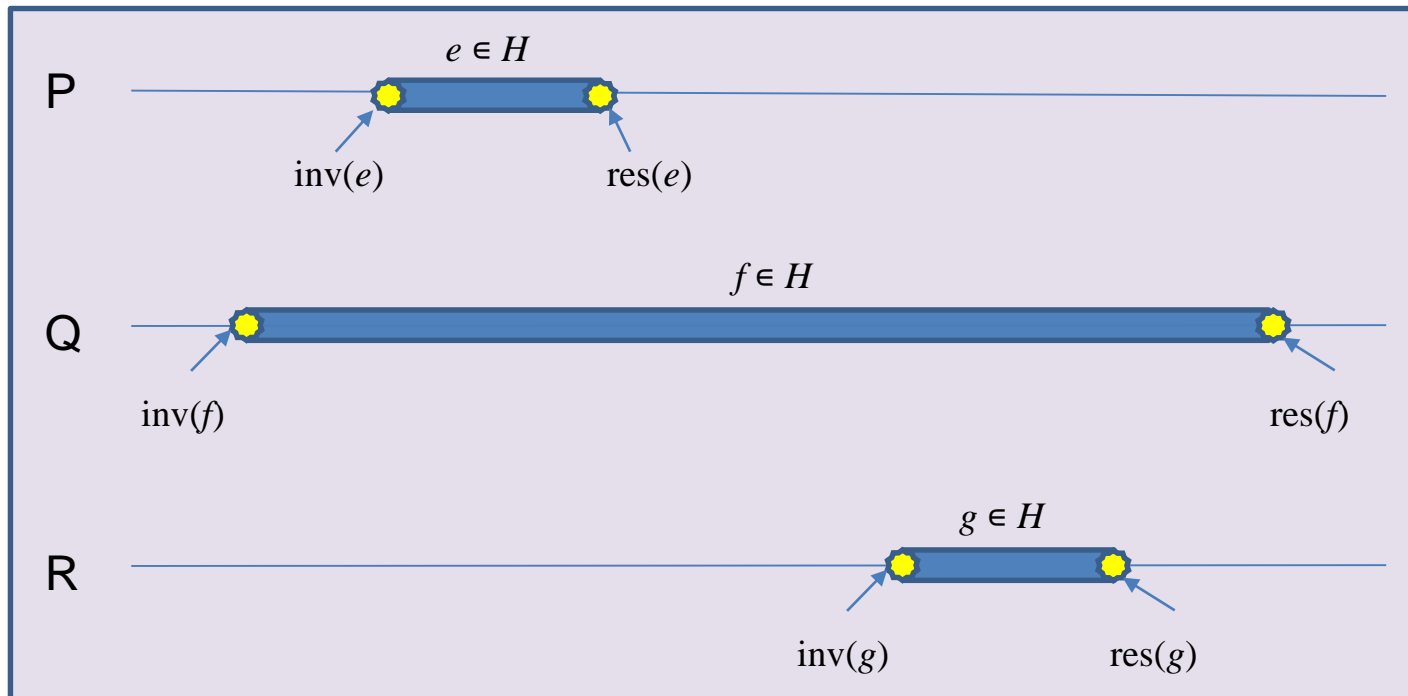
Линеаризуемо \Rightarrow есть точки линеаризации

- **Даны:**
 - Линеаризация операций $(L(H), \rightarrow_{L(H)})$
- **Найти:**
 - Точки линеаризации $p(e) \in G$ для всех $e \in H$
- **Очевидно получим**
 - ✓ Полный порядок над ними $\rightarrow_{L(P)}$ из условия согласованности:
$$\forall e, f \in H: e \rightarrow_{L(H)} f \Leftrightarrow p(e) \rightarrow_{L(P)} p(f)$$
 - ✓ $\rightarrow_{L(P)}$ удов. последовательной спецификации всех объектов

Так просто не получится!

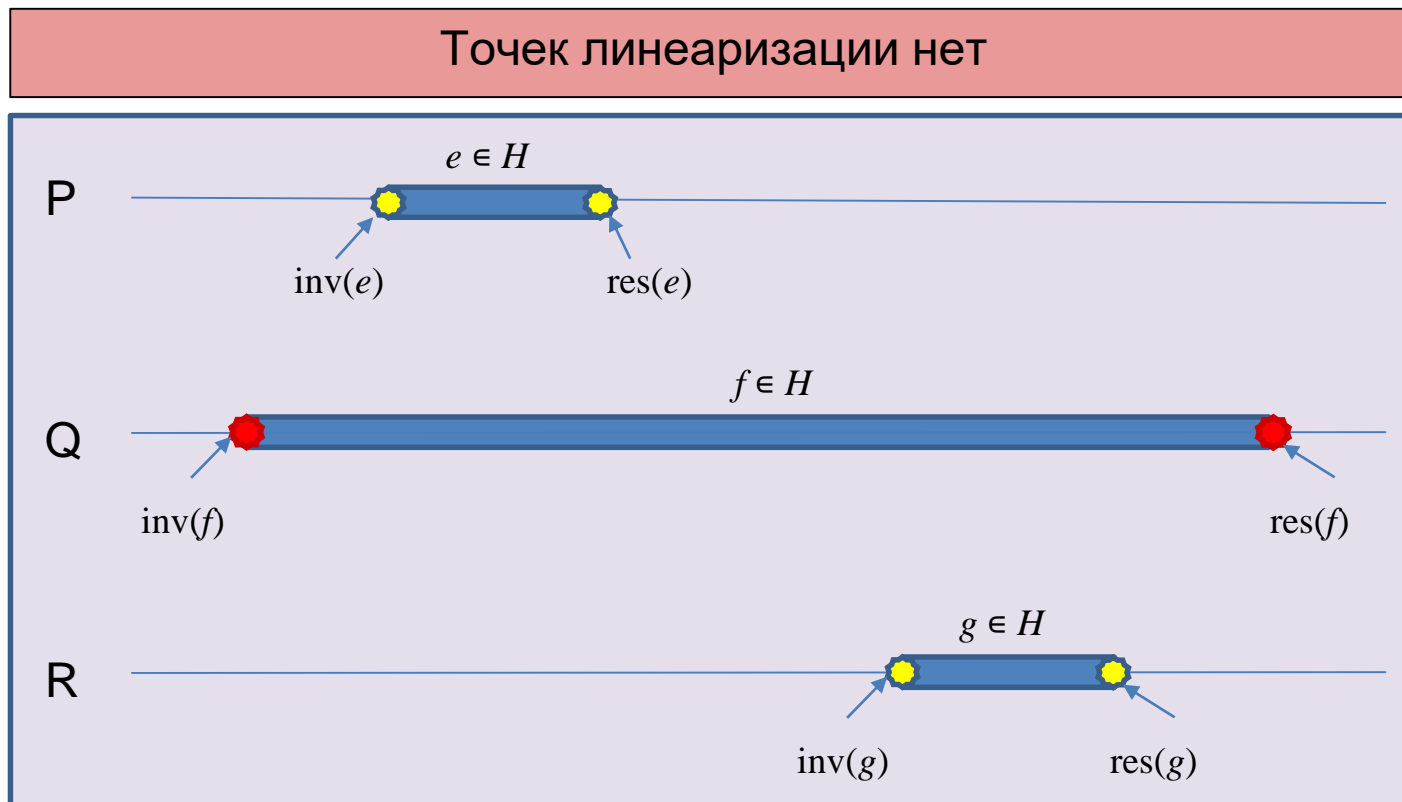
- Три **операции** e, f, g и отношение $e \rightarrow_H g$
- Шесть **событий** полностью упорядоченных (как на картинке)
- **Линеаризация операций** $e \rightarrow f \rightarrow g$

Корректная декомпозиция $(H, G, \rightarrow_G, \text{inv}, \text{res})$



Так просто не получится!

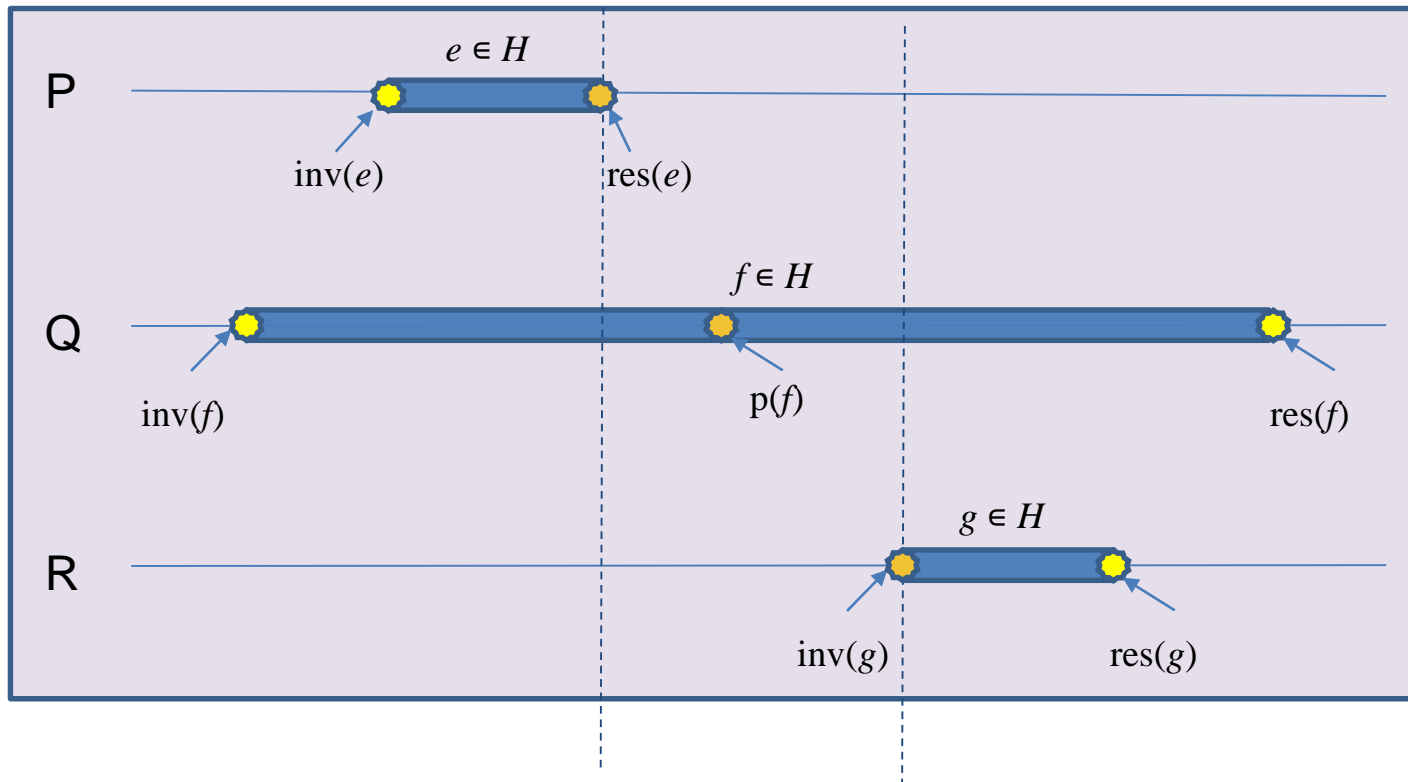
- Три операции e, f, g и отношение $e \rightarrow_H g$
- Шесть событий полностью упорядоченных (как на картинке)
- Линеаризация $e \rightarrow f \rightarrow g$



Так просто не получится!

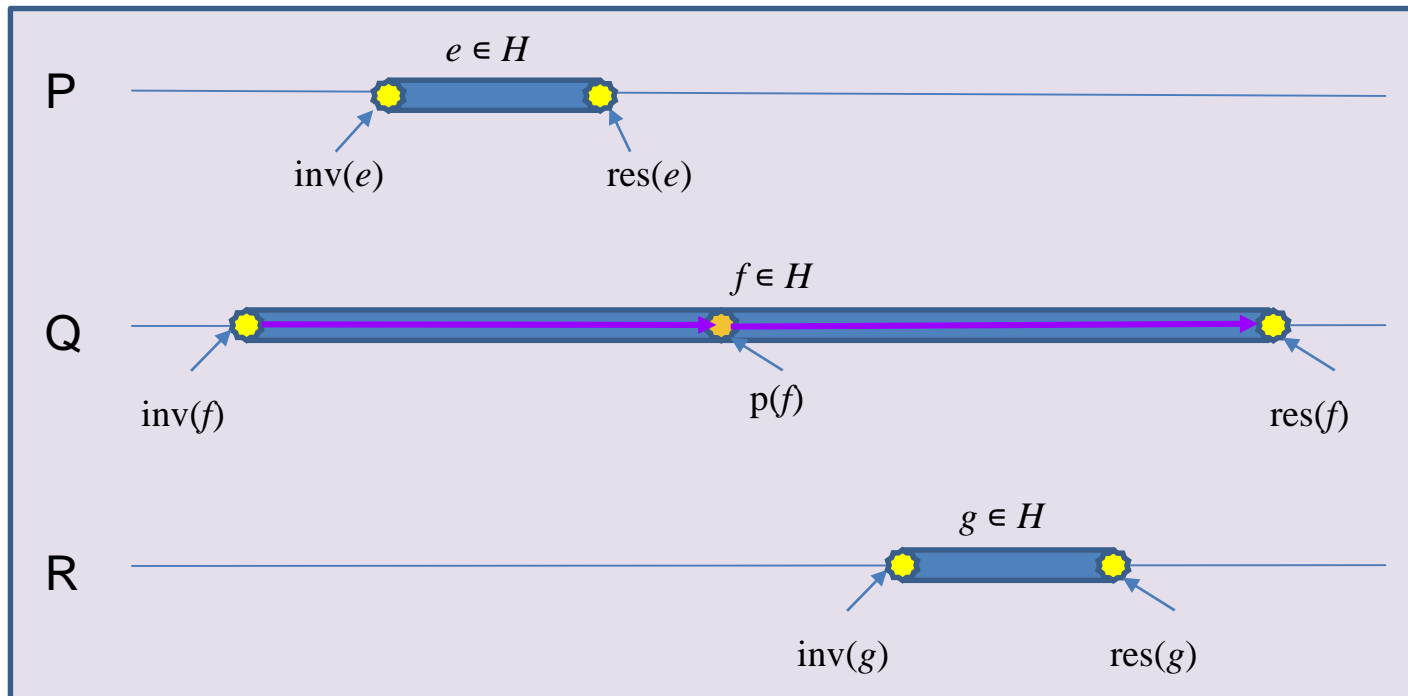
- Три **операции** e, f, g и отношение $e \rightarrow_H g$
- Шесть **событий** полностью упорядоченных (как на картинке)
- **Линеаризация** $e \rightarrow f \rightarrow g$

Но их можно доопределить



Определение точек линеаризации

- Добавим **события** $p(e) \in G$ для всех $e \in H$
- Введем на них порядок
 - Согласно с вызовом и завершением $\text{inv}(e) \rightarrow_G p(e) \rightarrow_G \text{res}(e)$
 - **Транзитивно замкнем**



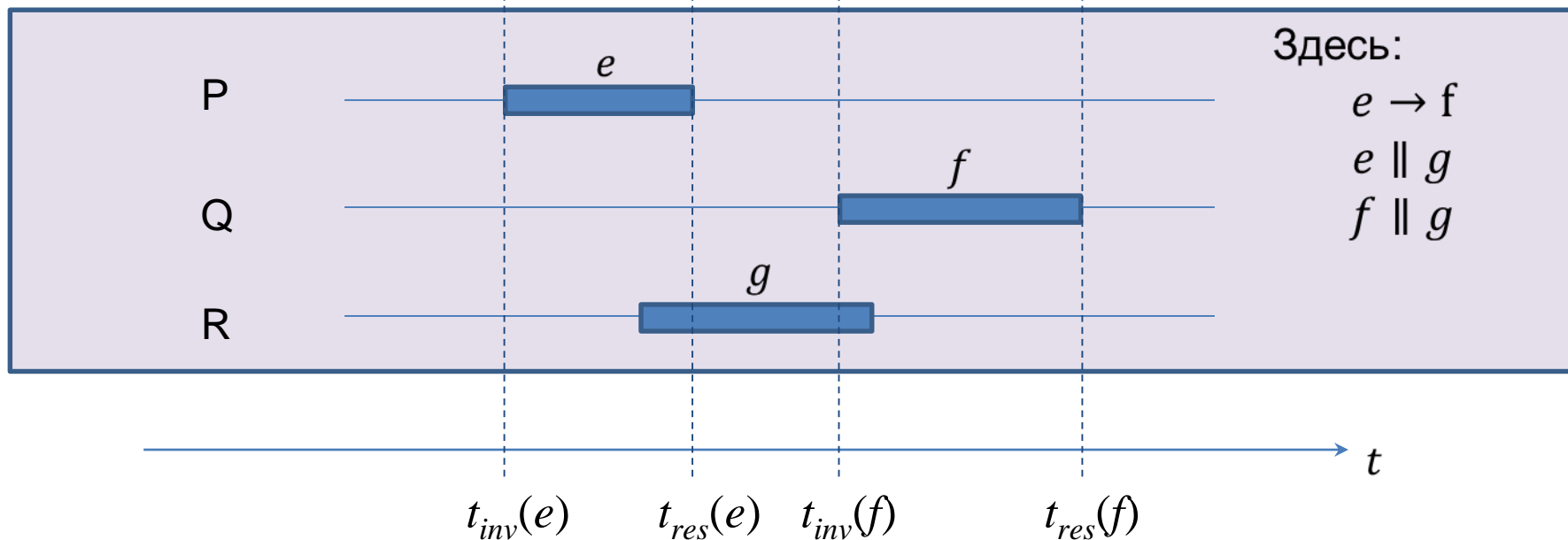
Определение точек линеаризации

- Добавим **события** $p(e) \in G$ для всех $e \in H$
- Введем на них порядок
 - Согласно с вызовом и завершением $\text{inv}(e) \rightarrow_G p(e) \rightarrow_G \text{res}(e)$
 - **Транзитивно замкнем**
- **Тогда очевидно**
 - По линеаризации всех операций $(L(H), \rightarrow_{L(H)})$
 - Найдем линеаризации этих событий $p(e)$

Модель глобального времени

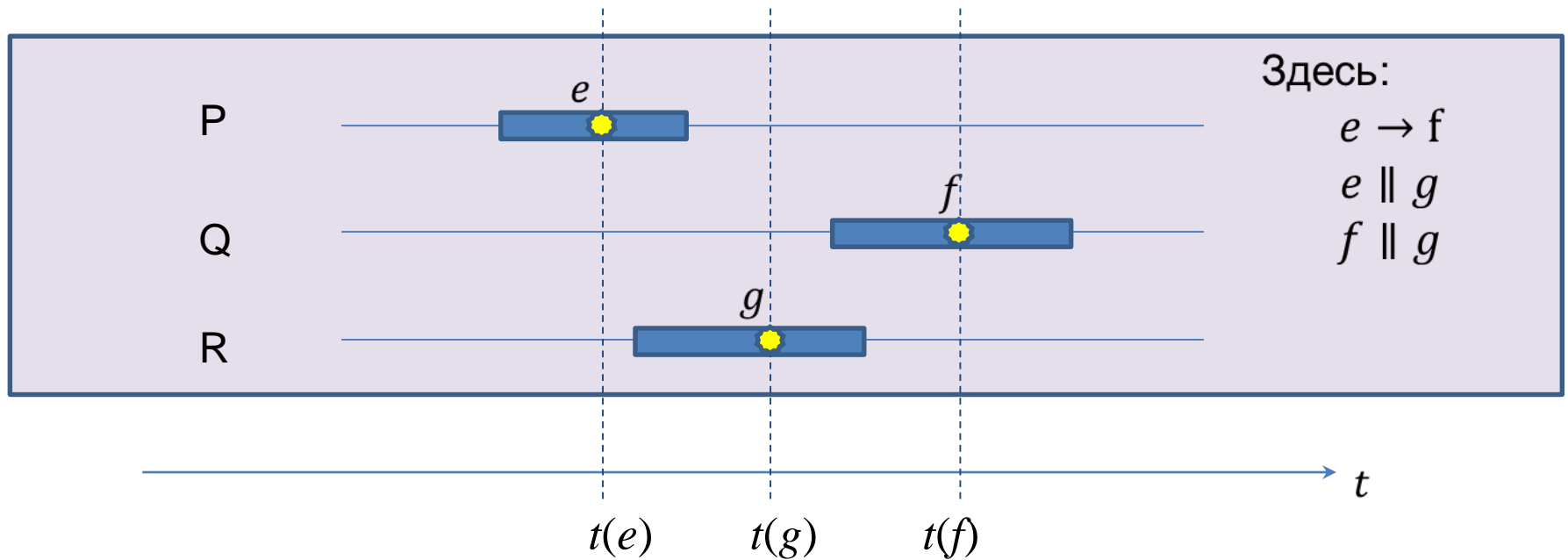
Частный случай декомпозиции!

- **События** G это пары $(\text{proc}, \text{time})$
- **Операция** имеет время начала и завершения $t_{\text{inv}}(e), t_{\text{res}}(e)$
 $\text{inv}(e) = (\text{proc}(e), t_{\text{inv}}(e)); \text{res}(e) = (\text{proc}(e), t_{\text{res}}(e))$
- **Декомпозиция** $e = \{(\text{proc}, \text{time}) : \text{proc} = \text{proc}(e), t_{\text{inv}}(e) \leq \text{time} \leq t_{\text{res}}(e)\}$



Модель глобального времени

- Точкам линеаризации
 - всегда можно сопоставить конкретное время $t(e)$
 - найти событие $p(e) = (\text{proc}(e), t(e))$
- В глобальном времени теорема верна в обе стороны



Основные выводы и следствия

- Чтобы доказать линейризуемость **операции** *достаточно* предъявить точки линейризации в декомпозиции на уровень ниже
 - Но их может *не быть* и надо искать другие способы док-ва
- Если операции низкого уровня линейризуемы (атомарны), то мы получаем полный порядок над $inv(e)$ и $res(e)$ который нужен для построения локальной линейризуемости операций высокого уровня.

Дальше будем опускать буковки G и H и писать как $e \rightarrow f$ так и $res(e) \rightarrow inv(f)$, а какая именно операция «произошло до» используется будет понятно из контекста

Псевдокод и формализм

Программный порядок

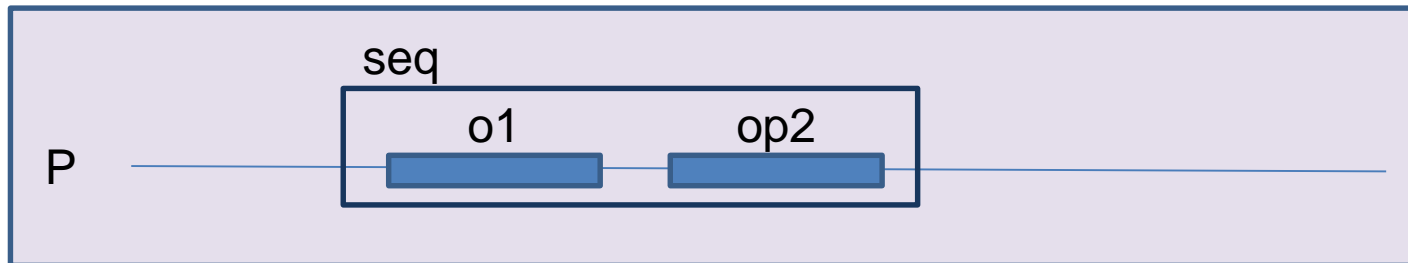
def seq:

1: *o1*

2: *op2*

Процедура **seq** последовательно выполняет ***o1*** а потом ***op2***

- Значит в любом исполнении этого кода будут выполняться свойство $op1 \rightarrow op2 \equiv res(op1) \rightarrow inv(op2)$
 - А значит $inv(op1) \rightarrow res(op1) \rightarrow inv(op2) \rightarrow res(op2)$
- Определим вызов и завершение составной операции **seq** как
 - $inv(seq) \stackrel{\text{def}}{=} inv(op1)$ и $res(seq) \stackrel{\text{def}}{=} res(op2)$
 - Это позволит ввести отношение «произошло до» на составных операциях, и далее делать из них более сложные операции



СЛОЖНЫЙ КОД

```
class Stack:  
    int top // init 0  
    int a[] // indexed from 0  
  
    def push(x):  
        1: curTop = top  
        2: a[curTop] = x  
        3: top = curTop + 1  
  
    def pop:  
        4: curTop = top  
        5: result = a[curTop - 1]  
        6: top = curTop - 1
```

- Это наш псевдо-код, где каждая отдельная операция-строка **атомарна**
- Поэтому анализируем через чередование

СЛОЖНЫЙ КОД

class Stack:

int top // init 0

int a[] // indexed from 0

def push(x):

1: curTop = top

2: a[curTop] = x

3: top = curTop + 1

def pop:

4: curTop = top

5: *result* = a[curTop - 1]

6: top = curTop - 1

top = 0; a = []

Thread 0:

Thread 1:

СЛОЖНЫЙ КОД

class Stack:

int top // init 0

int a[] // indexed from 0

def push(x):

1: curTop = top

2: a[curTop] = x

3: top = curTop + 1

def pop:

4: curTop = top

5: *result* = a[curTop - 1]

6: top = curTop - 1

top = 1; a = [1]

Thread 0: push(1)

curTop = 0

a[0] = 1

top = 1

Thread 1:

СЛОЖНЫЙ КОД

class Stack:

int top // init 0

int a[] // indexed from 0

def push(x):

1: curTop = top

2: a[curTop] = x

3: top = curTop + 1

def pop:

4: curTop = top

5: *result* = a[curTop - 1]

6: top = curTop - 1

top = 2; a = [1, 2]

Thread 0: push(1)

curTop = 0

a[0] = 1

top = 1

Thread 1: push(2)

curTop = 1

a[1] = 2

top = 2

СЛОЖНЫЙ КОД

class Stack:

int top // init 0

int a[] // indexed from 0

def push(x):

1: curTop = top

2: a[curTop] = x

3: top = curTop + 1

def pop:

4: curTop = top

5: *result* = a[curTop - 1]

6: top = curTop - 1

top = 0; a = []

Thread 0: push(1)

Thread 1: push(2)

СЛОЖНЫЙ КОД

class Stack:

int top // init 0

int a[] // indexed from 0

def push(x):

1: curTop = top

2: a[curTop] = x

3: top = curTop + 1

def pop:

4: curTop = top

5: *result* = a[curTop - 1]

6: top = curTop - 1

top = 0; a = []

Thread 0: push(1)

curTop = 0

Thread 1: push(2)

СЛОЖНЫЙ КОД

class Stack:

int top // init 0

int a[] // indexed from 0

def push(x):

1: curTop = top

2: a[curTop] = x

3: top = curTop + 1

def pop:

4: curTop = top

5: *result* = a[curTop - 1]

6: top = curTop - 1

top = 0; a = []

Thread 0: push(1)

curTop = 0

Thread 1: push(2)

curTop = 0

СЛОЖНЫЙ КОД

class Stack:

int top // init 0

int a[] // indexed from 0

def push(x):

1: curTop = top

2: **a[curTop] = x**

3: top = curTop + 1

def pop:

4: curTop = top

5: *result* = a[curTop - 1]

6: top = curTop - 1

top = 0; a = [1]

Thread 0: push(1)

curTop = 0

a[0] = 1

Thread 1: push(2)

curTop = 0

СЛОЖНЫЙ КОД

class Stack:

int top // init 0

int a[] // indexed from 0

def push(x):

1: curTop = top

2: a[curTop] = x

3: top = curTop + 1

def pop:

4: curTop = top

5: *result* = a[curTop - 1]

6: top = curTop - 1

top = 0; a = [2]

Thread 0: push(1)

curTop = 0

a[0] = 1

Thread 1: push(2)

curTop = 0

a[0] = 2

СЛОЖНЫЙ КОД

class Stack:

int top // init 0

int a[] // indexed from 0

def push(x):

1: curTop = top

2: a[curTop] = x

3: top = curTop + 1

def pop:

4: curTop = top

5: *result* = a[curTop - 1]

6: top = curTop - 1

top = 1; a = [2]

Thread 0: push(1)

curTop = 0

a[0] = 1

top = 1

Thread 1: push(2)

curTop = 0

a[0] = 2

СЛОЖНЫЙ КОД

class Stack:

int top // init 0

int a[] // indexed from 0

def push(x):

1: curTop = top

2: a[curTop] = x

3: top = curTop + 1

def pop:

4: curTop = top

5: *result* = a[curTop - 1]

6: top = curTop - 1

top = 1; a = [2]

Thread 0: push(1)

curTop = 0

a[0] = 1

top = 1

Thread 1: push(2)

curTop = 0

a[0] = 2

top = 1

СЛОЖНЫЙ КОД

class Stack:

```
int top // init 0  
int a[] // indexed from 0
```

def push(x):

```
1: curTop = top  
2: a[curTop] = x  
3: top = curTop + 1
```

def pop:

```
4: curTop = top  
5: result = a[curTop - 1]  
6: top = curTop - 1
```

top = 1; a = [2]

Thread 0: push(1)

```
curTop = 0  
a[0] = 1  
top = 1
```

Thread 1: push(2)

```
curTop = 0  
a[0] = 2  
top = 1
```

Потеряли **push(1)**

СЛОЖНЫЙ КОД

```
class Stack:  
    int top // init 0  
    int a[] // indexed from 0  
  
    def push(x):  
        1: curTop = top  
        2: a[curTop] = x  
        3: top = curTop + 1  
  
    def pop:  
        4: curTop = top  
        5: result = a[curTop - 1]  
        6: top = curTop - 1
```

**Не безопасно использовать из
разных потоков (not thread-safe)**

Нет линейризуемости!

ПОСТРОЕНИЕ ЛИНЕАРИЗУЕМЫХ ОБЪЕКТОВ

Построение линеаризуемых объектов

- **Как** построить линеаризуемый объект из последовательного?
 - ❖ *Если у нас есть атомарные регистры как аппаратные примитивы*

```
class Stack:  
    int top // init 0  
    int a[] // indexed from 0  
  
    def push(x):  
        1: curTop = top  
        2: a[curTop] = x  
        3: top = curTop + 1  
  
    def pop:  
        4: curTop = top  
        5: result = a[curTop - 1]  
        6: top = curTop - 1
```

Операция «потерялась» из-за
параллельности

**Как предотвратить эту
параллельность?**

Блокировки (Locks)



Взаимное исключение

- Защитим каждую операцию специальным объектом **mutex** (mutual exclusion), также известного как **блокировка (lock)**

```
class Stack:
    Mutex mutex
    ...

    def push(x):
        mutex.lock
        ...
        mutex.unlock

    def pop:
        mutex.lock
        ...
        mutex.unlock
```

Так чтобы реализация (тело) каждого метода не выполнялось **одновременно**, то есть все тела методов выполнялись бы последовательно.

Значит будет работать обычный последовательный код

Взаимное исключение формально

Протокол

```
thread Pid:  
  loop forever:  
    1: nonCriticalSection  
    2: mutex.lock  
    3: criticalSection  
    4: mutex.unlock
```

- Главное свойство называется **взаимное исключение**.
- **X1**: Критические секции не могут выполняться параллельно:

$$\forall i, j : i \neq j \Rightarrow CS_i \rightarrow CS_j \vee CS_j \rightarrow CS_i$$

- Это значит, что выполнение критических секций будет **линеаризуемо**
- Это требование **корректности** протокола взаимной блокировки

Взаимное исключение, попытка 1

shared boolean want

def lock:

```
1: while want:  
2:   pass // wait  
3: want = true
```

def unlock:

```
4: want = false
```

thread P_{id}:

```
loop forever:  
5: nonCS  
6: lock  
7: CS  
8: unlock
```

Взаимное исключение, попытка 1

shared boolean want

def lock:

```
1: while want:  
2:   pass // wait  
3: want = true
```

def unlock:

```
4: want = false
```

thread P_{id}:

```
loop forever:  
5: nonCS  
6: lock  
7: CS  
8: unlock
```

want = false

Thread 0:

Thread 1:

Взаимное исключение, попытка 1

shared boolean want

def lock:

```
1: while want:
2:   pass // wait
3: want = true
```

def unlock:

```
4: want = false
```

thread P_{id}:

```
loop forever:
5: nonCS
6: lock
7: CS
8: unlock
```

want = false

Thread 0:

```
1: rd want == false
```

Thread 1:

Взаимное исключение, попытка 1

shared boolean want

def lock:

```
1: while want:
2:   pass // wait
3: want = true
```

def unlock:

```
4: want = false
```

thread P_{id}:

```
loop forever:
5: nonCS
6: lock
7: CS
8: unlock
```

want = false

Thread 0:

```
1: rd want == false
```

Thread 1:

```
1: rd want == false
```

Взаимное исключение, попытка 1

shared boolean want

def lock:

```
1: while want:  
2:   pass // wait  
3:   want = true
```

def unlock:

```
4: want = false
```

thread P_{id}:

```
loop forever:  
5: nonCS  
6: lock  
7: CS  
8: unlock
```

want = true

Thread 0:

```
1: rd want == false  
  
3: wr want := true
```

Thread 1:

```
1: rd want == false
```

Взаимное исключение, попытка 1

shared boolean want

def lock:

```
1: while want:  
2:   pass // wait  
3: want = true
```

def unlock:

```
4: want = false
```

thread P_{id}:

```
loop forever:  
5: nonCS  
6: lock  
7: CS  
8: unlock
```

want = true

Thread 0:

```
1: rd want == false  
  
3: wr want := true
```

Thread 1:

```
1: rd want == false  
  
3: wr want := true
```


Взаимное исключение, попытка 1

shared boolean want

def lock:

```
1: while want:  
2:   pass // wait  
3: want = true
```

def unlock:

```
4: want = false
```

thread P_{id}:

```
loop forever:  
5: nonCS  
6: lock  
7: CS  
8: unlock
```

want = true

Thread 0:

```
1: rd want == false  
  
3: wr want := true  
  
7: CS
```

Thread 1:

```
1: rd want == false  
  
3: wr want := true
```

Взаимное исключение, попытка 1

shared boolean want

def lock:

```
1: while want:  
2:   pass // wait  
3: want = true
```

def unlock:

```
4: want = false
```

thread P_{id}:

```
loop forever:  
5: nonCS  
6: lock  
7: CS  
8: unlock
```

want = true

Thread 0:

```
1: rd want == false  
  
3: wr want := true  
  
7: CS
```

Thread 1:

```
1: rd want == false  
  
3: wr want := true  
  
7: CS
```

Взаимное исключение, попытка 1

shared boolean want

def lock:

```
1: while want:  
2:   pass // wait  
3: want = true
```

def unlock:

```
4: want = false
```

- Этот протокол **не гарантирует взаимное исключение**
 - Оба потока могут оказаться в критической секции **одновременно**

Для опровержения любого **свойства** достаточно продемонстрировать *одно* исполнение, которое его нарушает

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

def unlock:

```
4: want[id] = false
```

Взаимное исключение, попытка 2

Докажем взаимное исключение

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

<CS>

def unlock:

```
4: want[id] = false
```

Взаимное исключение, попытка 2

Докажем взаимное исключение

- Используя чередование
- От противного

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

<CS>

def unlock:

```
4: want[id] = false
```

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

```
def lock:
```

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

<CS>

```
def unlock:
```

```
4: want[id] = false
```

Докажем взаимное исключение

- Используя чередование
- От противного

Дано: Отрицание

Надо: Прийти к противоречию

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

```
def lock:
```

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

<CS>

```
def unlock:
```

```
4: want[id] = false
```

Докажем взаимное исключение

- Используя чередование
- От противного

Дано: Два потока *одновременно* в CS

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

<CS>

def unlock:

```
4: want[id] = false
```

Докажем взаимное исключение

- Используя чередование
- От противного

Дано: Два потока *одновременно* в CS

Значит: Один поток **id** зашел в CS
последним, в то время как другой
поток **1 - id** уже *был* в CS

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

```
def lock:
```

```
1: want[id] = true
```

```
2: while want[1 - id]:
```

```
3:    pass
```

<CS>

```
def unlock:
```

```
4: want[id] = false
```

Докажем взаимное исключение

- Используя чередование
- От противного

Дано: Два потока *одновременно* в CS

Значит: Один поток **id** зашел в CS *последним*, в то время как другой поток **1 - id** уже *был* в CS

Но: Зайти в CS можно только после
rd want[1 - id] == false

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

```
def lock:
```

```
1: want[id] = true
```

```
2: while want[1 - id]:
```

```
3:    pass
```

<CS>

```
def unlock:
```

```
4: want[id] = false
```

Докажем взаимное исключение

- Используя чередование
- От противного

Дано: Два потока *одновременно* в CS

Значит: Один поток **id** зашел в CS
последним, в то время как другой
поток **1 - id** уже *был* в CS

Но: Зайти в CS можно только после
rd want[1 - id] == false

Противоречие с тем, что другой поток
в CS ибо это значит **want[1 - id] == true**

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

def unlock:

```
4: want[id] = false
```

want[0] = false; want[1] = false

Thread 0:

Thread 1:

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

def unlock:

```
4: want[id] = false
```

want[0] = true; want[1] = false

Thread 0:

1: wr want[0] := true

Thread 1:

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

def unlock:

```
4: want[id] = false
```

want[0] = true; want[1] = true

Thread 0:

1: wr want[0] := true

Thread 1:

1: wr want[1] := true

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

def unlock:

```
4: want[id] = false
```

want[0] = true; want[1] = true

Thread 0:

```
1: wr want[0] := true  
2: rd want[1] == true
```

Thread 1:

```
1: wr want[1] := true
```

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

def unlock:

```
4: want[id] = false
```

want[0] = true; want[1] = true

Thread 0:

```
1: wr want[0] := true  
2: rd want[1] == true
```

Thread 1:

```
1: wr want[1] := true  
2: rd want[0] == true
```


Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

def unlock:

```
4: want[id] = false
```

want[0] = true; want[1] = true

Thread 0:

```
1: wr want[0] := true  
2: rd want[1] == true  
3: pass
```

Thread 1:

```
1: wr want[1] := true  
2: rd want[0] == true
```

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

def unlock:

```
4: want[id] = false
```

want[0] = true; want[1] = true

Thread 0:

```
1: wr want[0] := true  
2: rd want[1] == true  
3: pass
```

Thread 1:

```
1: wr want[1] := true  
2: rd want[0] == true  
3: pass
```

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:   pass
```

def unlock:

```
4: want[id] = false
```

want[0] = true; want[1] = true

Thread 0:

```
1: wr want[0] := true  
2: rd want[1] == true  
3: pass  
2: rd want[1] == true
```

Thread 1:

```
1: wr want[1] := true  
2: rd want[0] == true  
3: pass
```

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

def unlock:

```
4: want[id] = false
```

want[0] = true; want[1] = true

Thread 0:

```
1: wr want[0] := true  
2: rd want[1] == true  
3: pass  
2: rd want[1] == true
```

Thread 1:

```
1: wr want[1] := true  
2: rd want[0] == true  
3: pass  
2: rd want[0] == true
```

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

def lock:

```
1: want[id] = true  
2: while want[1 - id]:  
3:   pass
```

def unlock:

```
4: want[id] = false
```

want[0] = true; want[1] = true

Thread 0:

1: wr want[0] := true

2: rd want[1] == true

3: pass

2: rd want[1] == true

<бесконечно>

Thread 1:

1: wr want[1] := true

2: rd want[0] == true

3: pass

2: rd want[0] == true

<бесконечно>

Взаимное исключение, попытка 2

```
threadlocal int    id // 0 or 1  
shared boolean want[2]
```

```
def lock:
```

```
1: want[id] = true  
2: while want[1 - id]:  
3:     pass
```

```
def unlock:
```

```
4: want[id] = false
```

- Этот протокол гарантирует **взаимное исключение**
- Но не нет никакой гарантии **прогресса**.
- **x2: Отсутствие взаимной блокировки (deadlock-freedom).** Если несколько потоков пытаются войти в критическую секцию, то хотя бы один из них должен войти в критическую секцию за конечное время. *

* При условии что критические секции выполняются за конечное время

Взаимное исключение, попытка 3

```
threadlocal int    id // 0 or 1  
shared int        victim
```

```
def lock:
```

```
1: victim = id
```

```
2: while victim == id:
```

```
3:   pass
```

```
def unlock:
```

```
4: pass
```

Взаимное исключение, попытка 3

Докажем взаимное исключение

- Используя чередование
- От противного

```
threadlocal int    id // 0 or 1  
shared int        victim
```

```
def lock:
```

```
1: victim = id  
2: while victim == id:  
3:     pass
```

<CS>

```
def unlock:
```

```
4: pass
```


Взаимное исключение, попытка 3

```
threadlocal int    id // 0 or 1  
shared int        victim
```

```
def lock:
```

```
1: victim = id
```

```
2: while victim == id:
```

```
3:    pass
```

<CS>

```
def unlock:
```

```
4: pass
```

Докажем взаимное исключение

- Используя чередование
- От противного

Дано: Два потока *одновременно* в CS

Но: Поток **id** может зайти в CS можно только после

rd victim != id

|
|

Взаимное исключение, попытка 3

```
threadlocal int    id // 0 or 1  
shared int        victim
```

```
def lock:
```

```
1: victim = id
```

```
2: while victim == id:
```

```
3:    pass
```

<CS>

```
def unlock:
```

```
4: pass
```

Докажем взаимное исключение

- Используя чередование
- От противного

Дано: Два потока *одновременно* в CS

Но: Поток **id** может зайти в CS можно только после

rd victim != id

Если оба потока в CS то:

victim == 0 & victim == 1

Противоречие

Взаимное исключение, попытка 3

Докажем отсутствие взаимной блокировки

```
threadlocal int    id // 0 or 1  
shared int         victim
```

```
def lock:
```

```
1: victim = id
```

```
2: while victim == id:
```

```
3:   pass
```

<CS>

```
def unlock:
```

```
4: pass
```

Взаимное исключение, попытка 3

Докажем отсутствие взаимной блокировки

Дано: Два потока *одновременно* крутятся в цикле

```
threadlocal int    id // 0 or 1  
shared int         victim
```

def lock:

```
1: victim = id
```

```
2: while victim == id:
```

```
3: pass
```

<CS>

def unlock:

```
4: pass
```

Взаимное исключение, попытка 3

```
threadlocal int    id // 0 or 1  
shared int        victim
```

```
def lock:
```

```
1: victim = id
```

```
2: while victim == id:
```

```
3:    pass
```

<CS>

```
def unlock:
```

```
4: pass
```

Докажем отсутствие взаимной блокировки

Дано: Два потока *одновременно* крутятся в цикле

Но: Крутиться в цикле можно только после

```
rd victim == id
```

Если оба потока там, то:

```
victim == 0 & victim == 1
```

Противоречие

Взаимное исключение, попытка 3

```
threadlocal int    id // 0 or 1  
shared int        victim
```

def lock:

```
1: victim = id  
2: while victim == id:  
3:   pass
```

def unlock:

```
4: pass
```

victim = 0

Thread 0:

```
1: wr victim := 0  
2: rd victim == 0  
3: pass  
2: rd victim == 0  
3: pass  
<бесконечно>
```

Thread 1:

<выполняет non-CS,
не вызывает lock>

Взаимное исключение, попытка 3

```
threadlocal int    id // 0 or 1  
shared int        victim
```

```
def lock:
```

```
1: victim = id  
2: while victim == id:  
3:     pass
```

```
def unlock:
```

```
4: pass
```

- Есть **взаимное исключение** и **прогресс**
- Но можем заходить только по очереди. Поток будет **голодать**.
- **ХЗ**: Потребуем **отсутствие голодания** (starvation-freedom). Если какой-то поток пытается войти в критическую секцию, то он войдет в критическую секцию за конечное время. *

* При условии что критические секции выполняются за конечное время

Сводка необходимых свойств

Условие корректности

1. **Взаимное исключение** (mutual exclusion)

Условия прогресса

1. **Отсутствие взаимных блокировок** (deadlock freedom)
2. **Отсутствие голодания** (starvation freedom)

формально это *более сильное* условие прогресса

Взаимное исключение, алгоритм Петерсона

```
threadlocal int    id // 0 or 1  
shared boolean want[2]  
shared int         victim
```

def lock:

```
1: want[id] = true  
2: victim = id  
3: while want[1-id] and  
4:     victim == id:  
5:     pass
```

def unlock:

```
6: want[id] = false
```

Взаимное исключение, алгоритм Петерсона

```
threadlocal int    id // 0 or 1  
shared boolean want[2]  
shared int        victim
```

def lock:

```
1: want[id] = true  
2: victim = id  
3: while want[1 - id] and  
4:     victim == id:  
5:     pass
```

def unlock:

```
6: want[id] = false
```

- Гарантирует **взаимное исключение, отсутствие взаимной блокировки и отсутствие голодания.**
- Не первый изобретенный (1981), но простейший алгоритм для двух потоков.

Взаимное исключение, алгоритм Петерсона

Докажем взаимное исключение

Дано: Два потока *одновременно* в CS

Значит: Один поток **id** зашел в CS
последним, в то время как другой поток
1 - id уже был в CS

Но: Зайти в CS можно только после
rd want[1 - id] == false **ИЛИ**
rd victim != id

```
threadlocal int    id // 0 or 1
shared boolean want[2]
shared int         victim
```

def lock:

```
1: want[id] = true
2: victim = id
3: while want[1 - id] and
4:     victim == id:
5:     pass
```

<CS>

def unlock:

```
6: want[id] = false
```

Взаимное исключение, алгоритм Петерсона

Докажем взаимное исключение

Дано: Два потока *одновременно* в CS

Значит: Один поток **id** зашел в CS
последним, в то время как другой поток
1 - id уже был в CS

Но: Зайти в CS можно только после
rd want[1 - id] == false **ИЛИ**
rd victim != id

Случай 1: **rd want[1 - id] == false**

Противоречие с тем, что другой поток
в CS ибо это значит **want[1 - id] == true**

```
threadlocal int    id // 0 or 1
shared boolean want[2]
shared int         victim
```

def lock:

```
1: want[id] = true
```

```
2: victim = id
```

```
3: while want[1 - id] and
```

```
4:     victim == id:
```

```
5:     pass
```

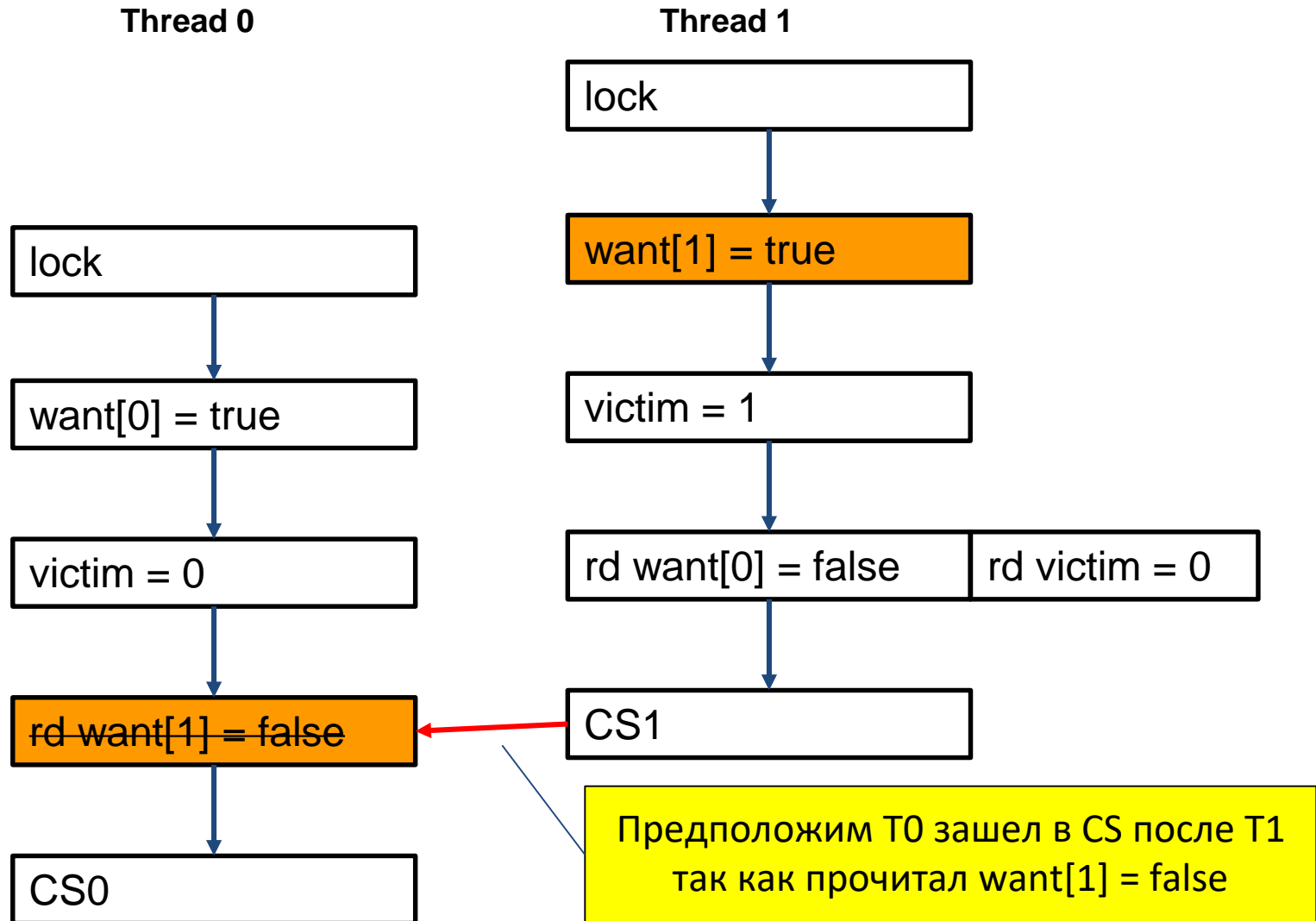
<CS>

def unlock:

```
6: want[id] = false
```

Взаимное исключение, алгоритм Петерсона

Случай 1



Взаимное исключение, алгоритм Петерсона

Докажем взаимное исключение

Дано: Два потока *одновременно* в CS

Значит: Один поток **id** зашел в CS
последним, в то время как другой поток
1 - id уже *был* в CS

Но: Зайти в CS можно только после
rd want[1 - id] == false **ИЛИ**
rd victim != id

Случай 2: **rd victim != id**

Но: Значит другой поток зашел по
want[id] = false, но того как наш поток
выполнил строку 1. **Противоречие!**

```
threadlocal int    id // 0 or 1
shared boolean want[2]
shared int         victim

def lock:
    1: want[id] = true
    2: victim = id
    3: while want[1 - id] and
    4:     victim == id:
    5:     pass
                                     <CS>

def unlock:
    6: want[id] = false
```

Взаимное исключение, алгоритм Петерсона

Случай 2

Thread 0

Thread 1

Должно быть так чтобы rd victim = 1

lock

want[0] = true

victim = 0

rd victim = 1

CS0

lock

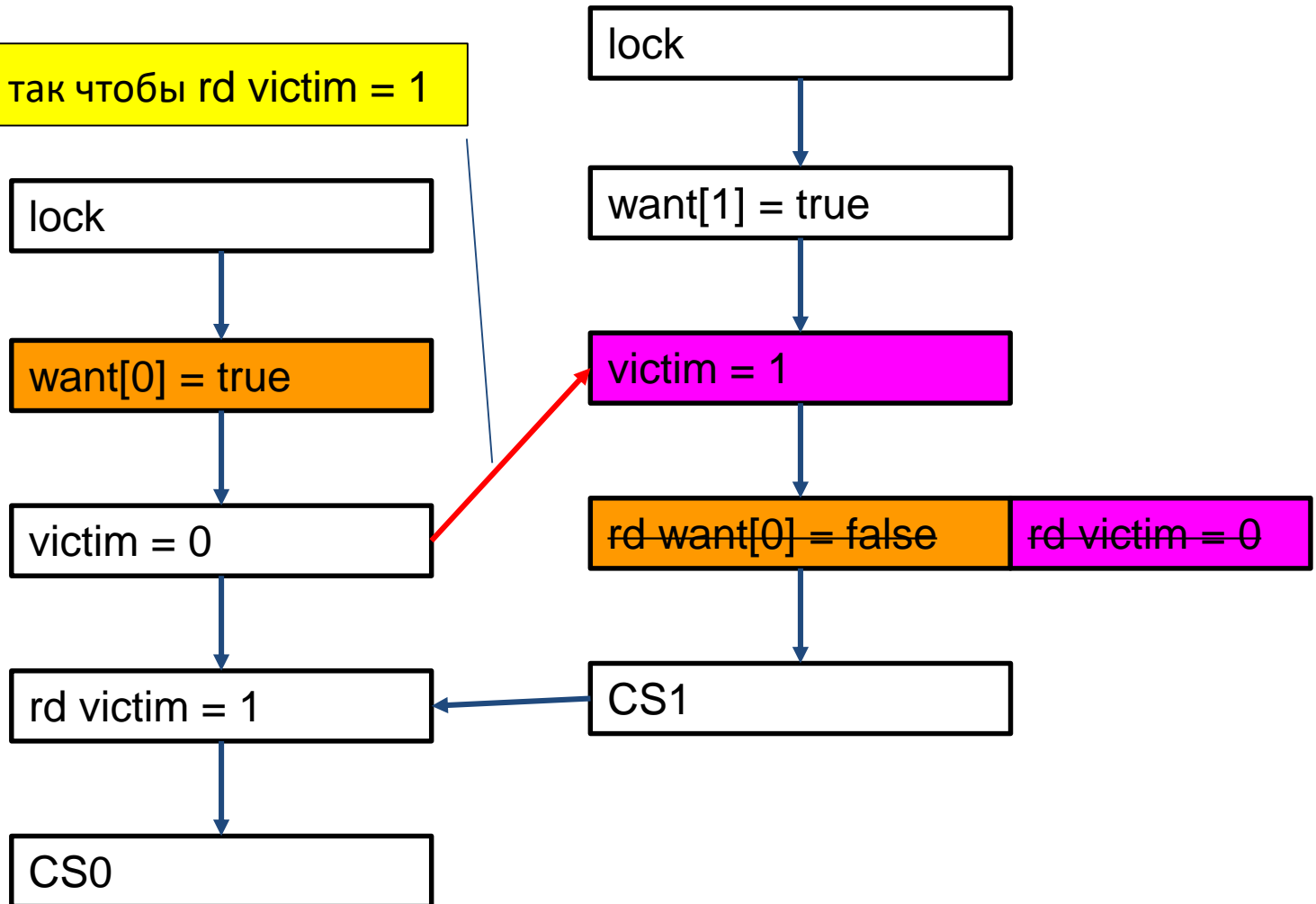
want[1] = true

victim = 1

rd want[0] = false

rd victim = 0

CS1



Взаимное исключение, алгоритм Петерсона

Важен порядок!!!

```
threadlocal int    id // 0 or 1
shared boolean want[2]
shared int         victim
```

def lock:

```
1: victim = id
```

```
2: want[id] = true
```

```
3: while want[1 - id] and
```

```
4:     victim == id:
```

```
5:     pass
```

<CS>

def unlock:

```
6: want[id] = false
```

Докажем взаимное исключение

Дано: Два потока *одновременно* в CS

Значит: Один поток **id** зашел в CS
последним, в то время как другой поток
1 - id уже *был* в CS

Но: Зайти в CS можно только после
rd want[1 - id] == false **ИЛИ**
rd victim != id

Случай 2: **rd victim != id**

Но: Значит другой поток зашел по
want[id] = false, но того как наш поток
выполнил строку 2. Но 1 мог уже! **Ой!**

Взаимное исключение, алгоритм Петерсона

Докажем отсутствие взаимной блокировки

Дано: Два потока *одновременно* крутятся в цикле

```
threadlocal int    id // 0 or 1
shared boolean want[2]
shared int         victim
```

def lock:

```
1: want[id] = true
2: victim = id
3: while want[1 - id] and
4:     victim == id:
5:     pass
```

def unlock:

```
6: want[id] = false
```

Взаимное исключение, алгоритм Петерсона

```
threadlocal int    id // 0 or 1
shared boolean want[2]
shared int         victim
```

def lock:

```
1: want[id] = true
2: victim = id
3: while want[1 - id] and
4:     victim == id:
5:     pass
```

def unlock:

```
6: want[id] = false
```

Докажем отсутствие взаимной блокировки

Дано: Два потока *одновременно* крутятся в цикле

Значит:

```
want[0] == true &
want[1] == true &
victim == 0 & victim == 1
```

Противоречие!

Взаимное исключение, алгоритм Петерсона

Докажем отсутствие голодания

Дано: Один поток пытается войти в критическую секцию

```
threadlocal int    id // 0 or 1
shared boolean want[2]
shared int         victim
```

def lock:

```
1: want[id] = true
2: victim = id
3: while want[1 - id] and
4:     victim == id:
5:     pass
```

def unlock:

```
6: want[id] = false
```

Взаимное исключение, алгоритм Петерсона

Докажем отсутствие голодания

Дано: Один поток пытается войти в критическую секцию

Очевидно получится!

Не может уйти в цикл из-за
`rd want[1 - id] == false`

```
threadlocal int    id // 0 or 1
shared boolean want[2]
shared int         victim
```

def lock:

```
1: want[id] = true
2: victim = id
3: while want[1 - id] and
4:     victim == id:
5:     pass
```

def unlock:

```
6: want[id] = false
```

Важное наблюдение

Для реализации взаимного исключения достаточно
иметь простейшие *общие объекты*:
атомарные регистры чтения / записи

Обобщение на N потоков

Взаимное исключение, алгоритм Петерсона для N потоков

```
threadlocal int    id // 0 to N-1
shared int         level[N]
shared int         victim[N]
```

def lock:

```
1: for j = 1..N-1:
2:   level[id] = j
3:   victim[j] = id
4:   while exist k: k != id and
5:     level[k] >= j and
6:     victim[j] == id:
7:     pass
```

def unlock:

```
8: level[id] = 0
```

Взаимное исключение, алгоритм Петерсона для N потоков

```
threadlocal int    id // 0 to N-1
shared int         level[N]
shared int         victim[N]
```

def lock:

```
1: for j = 1..N-1:
2:   level[id] = j
3:   victim[j] = id
4:   while exist k: k != id and
5:     level[k] >= j and
6:     victim[j] == id:
7:     pass
```

def unlock:

```
8: level[id] = 0
```

Для входа в CS надо пройти на N-1 уровней

Взаимное исключение, алгоритм Петерсона для N потоков

```
threadlocal int    id // 0 to N-1
shared int         level[N]
shared int         victim[N]
```

def lock:

```
1: for j = 1..N-1:
2:   level[id] = j
3:   victim[j] = id
4:   while exist k: k != id and
5:     level[k] >= j and
6:     victim[j] == id:
7:     pass
```

def unlock:

```
8: level[id] = 0
```

Обобщаем want на уровень j: $\text{level}[\text{id}] \geq j$

Взаимное исключение, алгоритм Петерсона для N потоков

```
threadlocal int    id // 0 to N-1
shared int         level[N]
shared int         victim[N]
```

def lock:

```
1: for j = 1..N-1:
2:   level[id] = j
3:   victim[j] = id
4:   while exist k: k != id and
5:     level[k] >= j and
6:     victim[j] == id:
7:     pass
```

def unlock:

```
8: level[id] = 0
```

Своя жертва на каждом уровне

Взаимное исключение, алгоритм Петерсона для N потоков

```
threadlocal int    id // 0 to N-1
shared int         level[N]
shared int         victim[N]
```

def lock:

```
1: for j = 1..N-1:
2:   level[id] = j
3:   victim[j] = id
4:   while exist k: k != id and
5:     level[k] >= j and
6:     victim[j] == id:
7:     pass
```

def unlock:

```
8: level[id] = 0
```

Для прохода на следующий уровень соревнуемся со всеми другими потоками

Взаимное исключение, алгоритм Петерсона для N потоков

```
threadlocal int    id // 0 to N-1
shared int         level[N]
shared int         victim[N]
```

def lock:

```
1: for j = 1..N-1:
2:   level[id] = j
3:   victim[j] = id
4:   while exist k: k != id and
5:     level[k] >= j and
6:     victim[j] == id:
7:     pass
```

def unlock:

```
8: level[id] = 0
```

- Гарантирует **взаимное исключение**, **отсутствие блокировки** и **отсутствие голодания**.

Взаимное исключение, алгоритм Петерсона для N потоков

```
threadlocal int    id // 0 to N-1
shared int         level[N]
shared int         victim[N]
```

def lock:

```
1: for j = 1..N-1:
2:   level[id] = j
3:   victim[j] = id
4:   while exist k: k != id and
5:     level[k] >= j and
6:     victim[j] == id:
7:     pass
```

def unlock:

```
8: level[id] = 0
```

Идея доказательства:

- При переходе с level на level+1 на level останется один поток - жертва в цикле **pass**

Взаимное исключение, алгоритм Петерсона для N потоков

```
threadlocal int    id // 0 to N-1
shared int         level[N]
shared int         victim[N]
```

def lock:

```
1: for j = 1..N-1:
2:   level[id] = j
3:   victim[j] = id
4:   while exist k: k != id and
5:     level[k] >= j and
6:     victim[j] == id:
7:     pass
```

def unlock:

```
8: level[id] = 0
```

Идея доказательства:

- При переходе с level на level+1 на level останется один поток - жертва в цикле **pass**

j = 1 - остался поток-жертва

j = 2 - остался поток-жертва

...

j = N-1 - остался поток-жертва

не более одного потока в CS

Взаимное исключение, алгоритм Петерсона для N потоков

```
threadlocal int    id // 0 to N-1
shared int         level[N]
shared int         victim[N]
```

def lock:

```
1: for j = 1..N-1:
2:   level[id] = j
3:   victim[j] = id
4:   while exist k: k != id and
5:     level[k] >= j and
6:     victim[j] == id:
7:     pass
```

def unlock:

```
8: level[id] = 0
```

- Гарантирует **взаимное исключение, отсутствие блокировки и отсутствие голодания.**
- Но не очень **честный**.
 - Невезучий поток может ждать пока другие потоки $O(N^2)$ раз войдут в критическую секцию (*квадратичное ожидание*)
 - Более честно было бы **линейное ожидание**

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1  
shared boolean want[N]    init false  
shared int        label[N]    init 0
```

def lock:

```
1: want[id] = true  
2: label[id] = max(label) + 1  
3: while exists k: k != id and  
4:     want[k] and  
5:     (label[k], k) < (label[id], id) :  
6:     pass
```

def unlock:

```
7: want[id] = false
```


Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1
shared boolean want[N]    init false
shared int         label[N]    init 0

def lock:
    1: want[id] = true
    2: label[id] = max(label) + 1 } doorway
    3: while exists k: k != id and
    4:     want[k] and
    5:     (label[k], k) < (label[id], id) :
    6:     pass

def unlock:
    7: want[id] = false
```

- Выдача номера очереди
- Обслуживание в порядке очереди

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1
shared boolean want[N]    init false
shared int         label[N]    init 0

def lock:
    1: want[id] = true
    2: label[id] = max(label) + 1 } doorway
    3: while exists k: k != id and
    4:     want[k] and
    5:     (label[k], k) < (label[id], id) :
    6:     pass

def unlock:
    7: want[id] = false
```

- Выдача номера очереди
- Обслуживание в порядке очереди

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1
shared boolean want[N]    init false
shared int         label[N]    init 0

def lock:
    1: want[id] = true
    2: label[id] = max(label) + 1 } doorway
    3: while exists k: k != id and
    4:     want[k] and
    5:     (label[k], k) < (label[id], id) :
    6:     pass

def unlock:
    7: want[id] = false
```

- Выдача номера очереди
- Обслуживание в порядке очереди
 - Упорядочивание по паре (label, k)

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1
shared boolean want[N]    init false
shared int         label[N]    init 0

def lock:
    1: want[id] = true
    2: label[id] = max(label) + 1 } doorway
    3: while exists k: k != id and
    4:     want[k] and
    5:     (label[k], k) < (label[id], id) :
    6:     pass

def unlock:
    7: want[id] = false
```

- Выдача номера очереди
- Обслуживание в порядке очереди
 - Упорядочивание по паре (label, k)
- **Ключевое свойство**
 - Если поток P выполнил doorway (DW) до потока Q, то у него более ранний номер очереди

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1
shared boolean want[N]    init false
shared int         label[N]    init 0

def lock:
  1: want[id] = true
  2: label[id] = max(label) + 1 } doorway
  3: while exists k: k != id and
  4:     want[k] and
  5:     (label[k], k) < (label[id], id) :
  6:   pass

                                     <CS>

def unlock:
  7: want[id] = false
```

**Докажем взаимное
исключение**

Дано: Как минимум два потока
одновременно в CS

Значит: Поток **id** зашел в CS
последним, в то время как другой
поток **k** != id уже *был* в CS

Но: Зайти в CS можно если
rd want[k] == false ИЛИ
rd (label[k], k) > (label[id], id)

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1
shared boolean want[N]    init false
shared int         label[N]    init 0

def lock:
    1: want[id] = true
    2: label[id] = max(label) + 1
    3: while exists k: k != id and
    4:     want[k] and
    5:     (label[k], k) < (label[id], id) :
    6:     pass
    <CS>

def unlock:
    7: want[id] = false
```

**Докажем взаимное
исключение**

Дано: Как минимум два потока
одновременно в CS

Значит: Поток **id** зашел в CS
последним, в то время как другой
поток **k** \neq id уже *был* в CS

Случай 1: **rd want[k] == false**

Противоречие!

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1
shared boolean want[N]    init false
shared int         label[N]    init 0

def lock:
  1: want[id] = true
  2: label[id] = max(label) + 1
  3: while exists k: k != id and
  4:     want[k] and
  5:     (label[k], k) < (label[id], id) :
  6:   pass
                                     <CS>

def unlock:
  7: want[id] = false
```

**Докажем взаимное
исключение**

Дано: Как минимум два потока
одновременно в CS

Значит: Поток **id** зашел в CS
последним, в то время как другой
поток **k** != id уже был в CS

Случай 2:
rd (label[k], k) >= (label[id], id)

Но: Значит другой поток зашел
по **want[id] == false** выполнив
свой doorway до потока **id**

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1
shared boolean want[N]    init false
shared int         label[N]    init 0

def lock:
    1: want[id] = true
    2: label[id] = max(label) + 1 } doorway
    3: while exists k: k != id and
    4:     want[k] and
    5:     (label[k], k) < (label[id], id) :
    6:     pass

def unlock:
    7: want[id] = false
```

- Есть корректность и все свойства прогресса:
 - Отсутствие взаимного исключение и голодания доказать не сложно

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1
shared boolean want[N]    init false
shared int         label[N]    init 0

def lock:
    1: want[id] = true
    2: label[id] = max(label) + 1 } doorway
    3: while exists k: k != id and
    4:     want[k] and
    5:     (label[k], k) < (label[id], id) :
    6:     pass

def unlock:
    7: want[id] = false
```

- Есть корректность и все свойства прогресса
- Обладает свойством **первый пришел, первый обслужен (FCFS)**
 - Это сильнее чем **линейное ожидание**
 - Самое сильное свойство прогресса

FCFS Формально

- Требование **First Come First Served** формализуется так
 - Метод **lock** должен состоять из двух последовательных секций

def lock:
1: **doorway**
2: **waiting**

- Секция **doorway** должна быть *wait free*, то есть выполняться за конечное число шагов независимо от действий других потоков
- Обозначим *исполнение* операций **doorway** как DW_i , а соответствующих операций **waiting** как WT_i .
- Секция **waiting** должна удовлетворять **FCFS** условию:
 - ❖ Если $DW_i \rightarrow DW_j$ (то есть $res(DW_i) \rightarrow inv(DW_j)$)
 - ❖ То $res(WT_i) \rightarrow res(WT_j)$
 - ✓ А при выполнении *взаимного исключения* $CS_i \rightarrow CS_j$

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int    id // 0 to N-1
shared boolean want[N]    init false
shared int         label[N]    init 0

def lock:
    1: want[id] = true
    2: label[id] = max(label) + 1 } doorway
    3: while exists k: k != id and
    4:     want[k] and
    5:     (label[k], k) < (label[id], id) :
    6:     pass

def unlock:
    7: want[id] = false
```


- Есть корректность и все свойства прогресса
- Обладает свойством **первый пришел, первый обслужен (FCFS)**
- Но метки должны быть бесконечными (их можно заменить на конечные метки)

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 2)

```
threadlocal int    id // 0 to N-1
shared boolean choosing[N] init false
shared int         label[N]      init inf

def lock:
  1: choosing[id] = true
  2: label[id] = max(label[id], inf) + 1
  3: choosing[id] = false
  4: while exists k: k != id and
  5:     (choosing[k] or
  6:     (label[k], k) < (label[id], id)) :
  7:   pass

def unlock:
  8: label[id] = inf
```



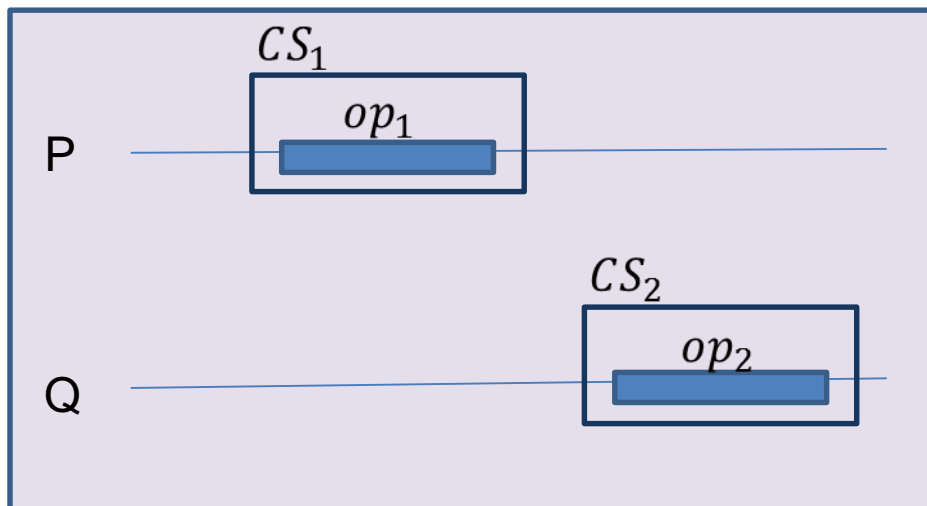
- Те же свойства
 - И метки тоже могут быть бесконечными, хотя мы их и сбрасываем при выходе из критической секции

Взаимное исключение (свод всех свойств)

- Алгоритм **взаимного исключения** должен иметь три свойства
 1. Взаимное исключение (**mutual exclusion**)
 2. Отсутствие взаимной блокировки (**deadlock freedom**)
 3. Отсутствие голодания (**starvation freedom**)
- Последнее свойство ничего не говорит о том, как долго может продолжаться ожидание (только то, что оно не будет вечным)
- Последнее свойство может быть последовательно усилено, превращаясь в условия честности (**fairness**)
 - a. Квадратичное ожидание (**quadratic wait**) – $O(N^2)$ операций
 - b. Линейное ожидание (**linear wait**) – $O(N)$ операций
 - c. Первый пришел, первый обслужен (**first come first served**)
 - ✓ Это самый честный из возможных вариантов, в простонародии называемый просто *честный* (**fair**)

Блокировка и корректная синхронизация

- Операции *защищенные блокировкой* (т.е. операции производимые в критической секции) не обязательно должны быть атомарными
- Из **взаимного исключения** $CS_1 \rightarrow CS_2$, т.е. $res(CS_1) \rightarrow inv(CS_2)$, но op_i работают внутри CS_i в том же потоке, значит $res(op_1) \rightarrow res(CS_1)$ и $inv(CS_2) \rightarrow inv(op_2)$, а значит из транзитивности $res(op_1) \rightarrow inv(op_2)$, что тоже самое что $op_1 \rightarrow op_2$



Операции op_1 и op_2 работают *последовательно* независимо от того, атомарны они сами или нет, а значит между ними не может быть **гонки данных (data race)**.
Отсюда же следует **линеаризуемость**

Блокировка на практике

Практичный test-and-set (TAS) spin lock

shared int locked

def lock:

1: **while** !locked.CAS(0, 1):

2: pass

def unlock:

3: locked = 0

Практичный test-and-set (TAS) spin lock

shared int locked

def lock:

1: **while** !locked.CAS(0, 1)

2: pass

def unlock:

3: locked = 0

Операция

**Compare-And-Set (CAS) aka
Test-And-Set (TAS)**

Практичный test-and-set (TAS) spin lock

Аппаратно в процессоре

shared int locked

def lock:

1: **while** !locked.CAS(0, 1)

2: pass

def unlock:

3: locked = 0

object register:

int value

def CAS(expect, update):

lock memory bus

if value == expect:

 value = update

result = true

else:

result = false

unlock memory bus

Практичный test-and-set (TAS) spin lock

shared int locked

def lock:

1: **while** !locked.CAS(0, 1)

2: pass

def unlock:

3: locked = 0

- Будет потреблять 100% CPU пока ждет
 - На практике немного spin, потом засыпаем через ОС

Практичный test-and-set (TAS) spin lock

```
shared int    locked

def lock:
1: while !locked.CAS(0, 1)
2:     pass

def unlock:
3: locked = 0
```

- Будет потреблять 100% CPU пока ждет
 - На практике немного spin, потом засыпаем через ОС
- Проблемы с масштабируемостью на реальном железе
- Подробности в отдельной лекции

Блокировка в Java

```
public class Stack {  
    public synchronized void enqueue(int x) {  
        // do something  
    }  
}
```

Блокировка в Java

```
public class Stack {  
    public void enqueue(int x) {  
        synchronized (this) {  
            // do something  
        }  
    }  
}
```

Блокировка в Java

```
public class Stack {  
    public void enqueue(int x) {  
        synchronized (this) { // MONITORENTER  
            // do something  
        } // MONITOREXIT  
    }  
}
```

Блокировка в Java

```
public class Stack {  
    private final Lock lock = new ReentrantLock();  
  
    public void enqueue(int x) {  
        lock.lock();  
        try {  
            // do something  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```


Блокировка в Kotlin

```
class Stack {  
    @Synchronized  
    fun enqueue(x: Int) {  
        // do something  
    }  
}
```

Блокировка в Kotlin

```
class Stack {  
    fun enqueue(x: Int) {  
        synchronized(this) {  
            // do something  
        }  
    }  
}
```

Блокировка в Kotlin

```
class Stack {  
    private val lock = ReentrantLock()  
  
    fun enqueue(x: Int) {  
        lock.lock()  
        try {  
            // do something  
        } finally {  
            lock.unlock()  
        }  
    }  
}
```

Блокировка в Kotlin

```
class Stack {  
    private val lock = ReentrantLock()  
  
    fun enqueue(x: Int) {  
        lock.withLock {  
            // do something  
        }  
    }  
}
```

Тонкая и двухфазная блокировки

Тонкая блокировка

```
class Stack:
```

```
    int top // init 0
```

```
    Mutex mTop // protects top
```

```
    int a[] // indexed from 0
```

```
    Mutex mA // protects a
```

Тонкая блокировка

```
class Stack
```

```
def push(x):
```

```
1: mTop.lock
```

```
2: curTop = top
```

```
3: top = curTop + 1
```

```
4: mTop.unlock
```

```
5: mA.lock
```

```
6: a[curTop] = x
```

```
7: mA.unlock
```

```
def pop:
```

```
8: mTop.lock
```

```
9: top = curTop - 1
```

```
10: curTop = top
```

```
11: mTop.unlock
```

```
12: mA.lock
```

```
13: result = a[curTop - 1]
```

```
14: mA.unlock
```

```
class Stack:
```

```
int top // init 0
```

```
Mutex mTop // protects top
```

```
int a[] // indexed from 0
```

```
Mutex mA // protects a
```

Тонкая блокировка

```
class Stack  
def push(x):  
    1: mTop.lock  
    2: curTop = top  
    3: top = curTop + 1  
    4: mTop.unlock  
    5: mA.lock  
    6: a[curTop] = x  
    7: mA.unlock  
  
def pop:  
    8: mTop.lock  
    9: curTop = top  
    10: top = curTop - 1  
    11: mTop.unlock  
    12: mA.lock  
    13: result = a[curTop - 1]  
    14: mA.unlock
```

top = 0; a = []

Thread 0: push(1)

Thread 1: pop()

Тонкая блокировка

```
class Stack
```

```
def push(x):
```

```
1: mTop.lock
```

```
2: curTop = top
```

```
3: top = curTop + 1
```

```
4: mTop.unlock
```

```
5: mA.lock
```

```
6: a[curTop] = x
```

```
7: mA.unlock
```

```
def pop:
```

```
8: mTop.lock
```

```
9: curTop = top
```

```
10: top = curTop - 1
```

```
11: mTop.unlock
```

```
12: mA.lock
```

```
13: result = a[curTop - 1]
```

```
14: mA.unlock
```

top = 1; a = []

Thread 0: push(1)

curTop = 0

top = 1

Thread 1: pop()

Тонкая блокировка

```
class Stack
```

```
  def push(x):
```

```
    1: mTop.lock  
    2: curTop = top  
    3: top = curTop + 1  
    4: mTop.unlock  
    5: mA.lock  
    6: a[curTop] = x  
    7: mA.unlock
```

```
  def pop:
```

```
    8: mTop.lock  
    9: curTop = top  
   10: top = curTop - 1  
   11: mTop.unlock  
   12: mA.lock  
   13: result = a[curTop - 1]  
   14: mA.unlock
```

top = 0; a = []

Thread 0: push(1)

curTop = 0
top = 1

Thread 1: pop()

curTop = 1
top = 0

Тонкая блокировка

```
class Stack  
def push(x):  
    1: mTop.lock  
    2: curTop = top  
    3: top = curTop + 1  
    4: mTop.unlock  
    5: mA.lock  
    6: a[curTop] = x  
    7: mA.unlock
```

```
def pop:  
    8: mTop.lock  
    9: curTop = top  
   10: top = curTop - 1  
   11: mTop.unlock  
   12: mA.lock  
   13: result = a[curTop - 1]  
   14: mA.unlock
```

top = 0; a = []

Thread 0: push(1)

curTop = 0
top = 1

Thread 1: pop()

curTop = 1
top = 0

result = ??????????

Тонкая блокировка

```
class Stack
```

```
def push(x):
```

```
    1: mTop.lock  
    2: curTop = top  
    3: top = curTop + 1  
    4: mTop.unlock  
    5: mA.lock  
    6: a[curTop] = x  
    7: mA.unlock
```

```
def pop:
```

```
    8: mTop.lock  
    9: curTop = top  
   10: top = curTop - 1  
   11: mTop.unlock  
   12: mA.lock  
   13: result = a[curTop - 1]  
   14: mA.unlock
```

top = 0; a = []

Thread 0: push(1)

curTop = 0
top = 1

Thread 1: pop()

curTop = 1
top = 0

result = ??????????


Не работает!

Двухфазная блокировка (2PL)

- Каждому общему объекту сопоставлена своя блокировка (тонкая блокировка)
- **Алгоритм 2-Phase Locking:**
 1. Взять блокировки на все необходимые объекты
 2. Выполнить операцию
 3. Отпустить все взятые блокировки
- Блокировки можно брать и отпускать в любом порядке

Двухфазная блокировка (2PL)

- Каждому общему объекту сопоставлена своя блокировка (тонкая блокировка)
- **Алгоритм 2-Phase Locking:**
 1. Взять блокировки на все необходимые объекты
 2. Выполнить операцию
 3. Отпустить все взятые блокировки
- Блокировки можно брать и отпускать в любом порядке



**2PL исполнение всегда
линеаризуемо**
(точка линеаризации между фазами)

Двухфазная блокировка: примеры

Обычно делают как-то так

def Proc1:

1: mutex1.lock

2: mutex2.lock

3: obj1.work

4: obj2.work

5: mutex2.unlock

6: mutex1.unlock

} lock

} unlock

Двухфазная блокировка: примеры

Обычно делают как-то так

def Proc1:

```
1: mutex1.lock  
2: mutex2.lock  
3: obj1.work  
4: obj2.work  
5: mutex2.unlock  
6: mutex1.unlock
```

} lock

} unlock

Можно оптимизировать

def Proc2:

```
1: mutex1.lock  
2: obj1.work  
3: mutex2.lock  
4: obj2.work  
5: mutex2.unlock  
6: mutex1.unlock
```

} lock

} unlock

Двухфазная блокировка: примеры

Обычно делают как-то так

def Proc1:

```
1: mutex1.lock  
2: mutex2.lock  
3: obj1.work  
4: obj2.work  
5: mutex2.unlock  
6: mutex1.unlock
```

} lock

} unlock

Порядок unlock не важен для 2PL

def Proc3:

```
1: mutex1.lock  
2: mutex2.lock  
3: obj1.work  
4: obj2.work  
5: mutex1.unlock  
6: mutex2.unlock
```

} lock

} unlock

Можно оптимизировать

def Proc2:

```
1: mutex1.lock  
2: obj1.work  
3: mutex2.lock  
4: obj2.work  
5: mutex2.unlock  
6: mutex1.unlock
```

} lock

} unlock

Двухфазная блокировка: примеры

Обычно делают как-то так

```
def Proc1:
  1: mutex1.lock
  2: mutex2.lock
  3: obj1.work
  4: obj2.work
  5: mutex2.unlock
  6: mutex1.unlock
```

lock

unlock

Порядок unlock не важен для 2PL

```
def Proc3:
  1: mutex1.lock
  2: mutex2.lock
  3: obj1.work
  4: obj2.work
  5: mutex1.unlock
  6: mutex2.unlock
```

lock

unlock

Можно оптимизировать

```
def Proc2:
  1: mutex1.lock
  2: obj1.work
  3: mutex2.lock
  4: obj2.work
  5: mutex2.unlock
  6: mutex1.unlock
```

lock

unlock

А так делать нельзя – нет 2PL

```
def Proc4:
  1: mutex1.lock
  2: obj1.work
  3: mutex1.unlock
  4: mutex2.lock
  5: obj2.work
  6: mutex2.unlock
```

Двухфазная блокировка: примеры

Обычно делают как-то так

def Proc1:

```
1: mutex1.lock  
2: mutex2.lock  
3: obj1.work  
4: obj2.work  
5: mutex2.unlock  
6: mutex1.unlock
```

} lock

} unlock

Порядок unlock не важен для 2PL

def Proc3:

```
1: mutex1.lock  
2: mutex2.lock  
3: obj1.work  
4: obj2.work  
5: mutex1.unlock  
6: mutex2.unlock
```

} lock

} unlock

Можно оптимизировать

def Proc2:

```
1: mutex1.lock  
2: obj1.work  
3: mutex2.lock  
4: obj2.work  
5: mutex2.unlock  
6: mutex1.unlock
```

} lock

} unlock

А так делать нельзя – нет 2PL

def Proc4:

```
1: mutex1.lock  
2: obj1.work  
3: mutex1.unlock  
4: mutex2.lock  
5: obj2.work  
6: mutex2.unlock
```

Использование взаимного исключения

Любой последовательный объект можно сделать параллельным, **линеаризуемым**:

- Грубая блокировка
 - Блокируем всю операцию целиком
- Тонкая блокировка
 - Блокируем только операции над **отдельными** внутренними объектами
 - Требуем двухфазной блокировки для обеспечения линеаризуемости

Проблема: взаимная блокировка

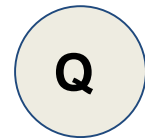
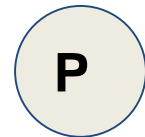
- При использовании блокировок можем получить взаимную блокировку потоков (deadlock)

Thread P	Thread Q
<pre>a.lock() b.lock() // do smth unlock(a, b)</pre>	<pre>b.lock() a.lock() // do smth unlock(a, b)</pre>

- Для обнаружения используют **граф ожидания (wait for graph)**
 - **Вершины = процессы (потоки)**
 - Ребро $P \rightarrow Q$ если процесс P ждет процесса Q

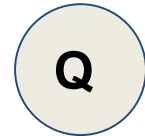
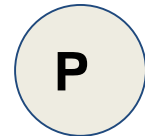
Граф ожидания

Thread P	Thread Q
<pre>a.lock() b.lock() // do smth unlock(a, b)</pre>	<pre>b.lock() a.lock() // do smth unlock(a, b)</pre>



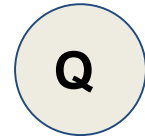
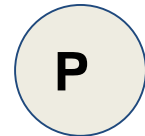
Граф ожидания

Thread P	Thread Q
<pre>1 a.lock() 3 b.lock() // do smth unlock(a, b)</pre>	<pre>2 b.lock() 4 a.lock() // do smth unlock(a, b)</pre>



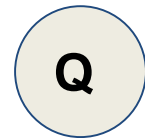
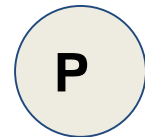
Граф ожидания

Thread P	Thread Q
<pre>1 a.lock() 3 b.lock() // do smth unlock(a, b)</pre>	<pre>2 b.lock() 4 a.lock() // do smth unlock(a, b)</pre>



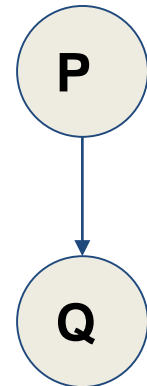
Граф ожидания

Thread P	Thread Q
<pre>1 a.lock() 3 b.lock() // do smth unlock(a, b)</pre>	<pre>2 b.lock() 4 a.lock() // do smth unlock(a, b)</pre>



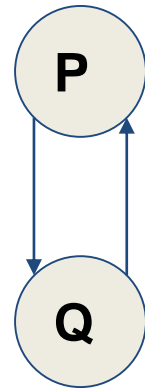
Граф ожидания

Thread P	Thread Q
<pre>1 a.lock() 3 b.lock() // do smth unlock(a, b)</pre>	<pre>2 b.lock() 4 a.lock() // do smth unlock(a, b)</pre>



Граф ожидания

Thread P	Thread Q
<pre>1 a.lock() 3 b.lock() // do smth unlock(a, b)</pre>	<pre>2 b.lock() 4 a.lock() // do smth unlock(a, b)</pre>



**Цикл в графе
ожидания**

Иерархическая блокировка

- Упорядочим все блокировки выстроив их в **иерархию**
- Всегда будем захватывать сначала более приоритетные блокировки до менее приоритетных
- Тогда взаимная блокировка невозможна
 - Почему!?

Итого: использование взаимного исключения

Любой последовательный объект можно сделать параллельным, **линеаризуемым**:

- Грубая блокировка
 - Блокируем всю операцию целиком
- Тонкая блокировка
 - Блокируем только операции над **отдельными** внутренними объектами
 - Требуем двухфазной блокировки для обеспечения линеаризуемости
 - Требуем иерархической блокировки для отсутствия взаимного исключения

Сводка проблем блокировки

- **Условные условия прогресса**
 - Гарантии только если критические секции выполняются за конечное время
- **Инверсия приоритетов**
 - Возникает при блокировке потоков с разным приоритетом на одном объекте
 - Обходятся путем поддержки блокировок на уровне ОС
- **Голодание**
 - Блокировка не достается низкоприоритетным потокам
 - Можно использовать честные блокировки
- **Последовательное выполнение операций**
 - Нет параллелизма при выполнении критической секции
 - \Rightarrow ограничена **вертикальная** масштабируемость