Concurrency & Clojure

Обычный подход к конкуррентности

- прямые ссылки на изменяемые структуры/данные
- надежда на правильные локи (как твои, так и чужие!)
  - правильные наборы локов
  - правильный порядок блокировки

Clojure-подход к конкурентности

Ссылки!

- нет способа расставлять блокировки вручную
- косвенные ссылки на неизменяемые (персистентные?) структуры данных
- принудительная семантика параллелизма для ссылок
- the ony thing that can be mutated in clojure
- единственные мутабельные объекты
- 4 вида: `var`, `atom`, `ref`, `agent`
  - "*unified succession model*"
- последние три поддерживают операцию `(deref reference)`

var

- хранение глобального состояния

```
(def a 1)                ;; => #'user/a
a                        ;; => 1
```

- возможен dynamic scope => может быть перезаписано

```
(def ^:dynamic b 10) ;; => #'user/b
b                    ;; => 10
(binding [b 7] b)    ;; => 7
(defn f [] b)        ;; => #'user/f
(f)                  ;; => 10
(binding [b 7] (f))  ;; => 7
```

- stack discipline

```
(binding [b 7] (+ (f) (binding [b 3] (f)))) ;; => 10
```

atom

- атомарная ячейка

```
(let [a (atom 10)] ...)
```

- позволяет обновлять состояние синхронно

```
(swap! a inc) ;; @a == 11
```

- но без координинации
- будет перезапущена, если свап не удался
- валидация предикатом:

```
(atom 100 :validator #(>= % 0))
```

`ref`

- синхронное управление состоянием, разделяется между тредами

```
(let [r (ref 17)] ...)
```

- может быть использована только внутри транзакций (STM)

```
(ref-set r 3)       ;; @r == 3 при удачной транзакции
(alter r (partial * 2)) ;; @r == 6
```

  - изменения атомарны и изолированы
  - спекулятивное выполнение, автоматический перезапуск
    - нужно избегать сайд-эффектов!

- коммутативные изменения (когда не важен порядок применения)

```
(commute r inc)   ;; @r == 7
```

- валидация предикатом как и в `atom`

`agent`

- асинхронное управление независимым состоянием, разделяется между тредами
- ячейка, хранящая состояние

```
(let [a (agent 9)] ...)
```

- очередь функций, **последовательно** мутирующих это состояние в отдельном тредпуле
  - fire & forget, мутация будет выполнена в будущем
  - или `await` для ожидания выполнения действия
- состояние прозрачно доступно в любой момент: `deref` / `@`

`agent`

- посылка сообщений за счёт
  - `send` для неблокирующих операций (cpu?)
  - `send-off` для блокирующих операций (io?)
  - `send-via` для указания собственного тредпула
- отправка собщений другим агентам произойдёт только после выполнения текущего действия
- координация внутри транзакции: агенты -- единственный способ выполнять сайдэффекты
  - отправка сообщений агентам будет выполена только после коммита

STM

- окружаем код `(dosync ...)` для объявления транзакции
  - защита ссылки от изменения другмим транзакциями: `ensure`
  - динамический контекст => вложенные транзакции поглощаются объемлющей
- на входе в транзакцию создаём снапшот
- единственная запись на выходе из транзакции
  - читатели не блокируют других читателей или писателей
  - писатели не блокируют читателей
- поддержка коммутативных операций (`commute` на `ref`)

Для всех ли задач требуется транзакционный доступ к ресурсам?
Есть ли другие варианты?

`DataFlow` - задачи:

- получаем запрос, достаём данные
- идём в базу
- спрашиваем у нескольких сторонних сервисов
- на основе результатов принимаем решение
- возвращаем ответ на запрос

ветвления?

Объекты плохо композируются

- кто угодно может выполнить метод объекта
  - потенциально в разных потоках, одновременно?
- "марионетки":
  - не объект решает что, когда и как ему делать, а его вызывающая сторона
    - не собака гавкнула, а мы гавкнули собакой

Чистых функций мало для композиции

- необходимо менять состояние в ходе работы системы
- или взаимодействовать с внешними системами

Популярное решение: события и коллбеки

- прямая связь компонент (например, прямой вызов функций) провоцирует проблемы
- UI: callback hell
- взаимодействие со сторонними системами: callback hell

Clojure-way: очереди?

- Плюсы:
  - разрывают прямую связь между producer и consumer
  - допускают произвольное количество producer/consumer
- Минусы:
  - блокируют реальные потоки (`java.util.concurrent` / `System.Collections.Concurrent`)
  - реальные потоки дороги:
    - время создания и переключения
    - память (для хранения стека)

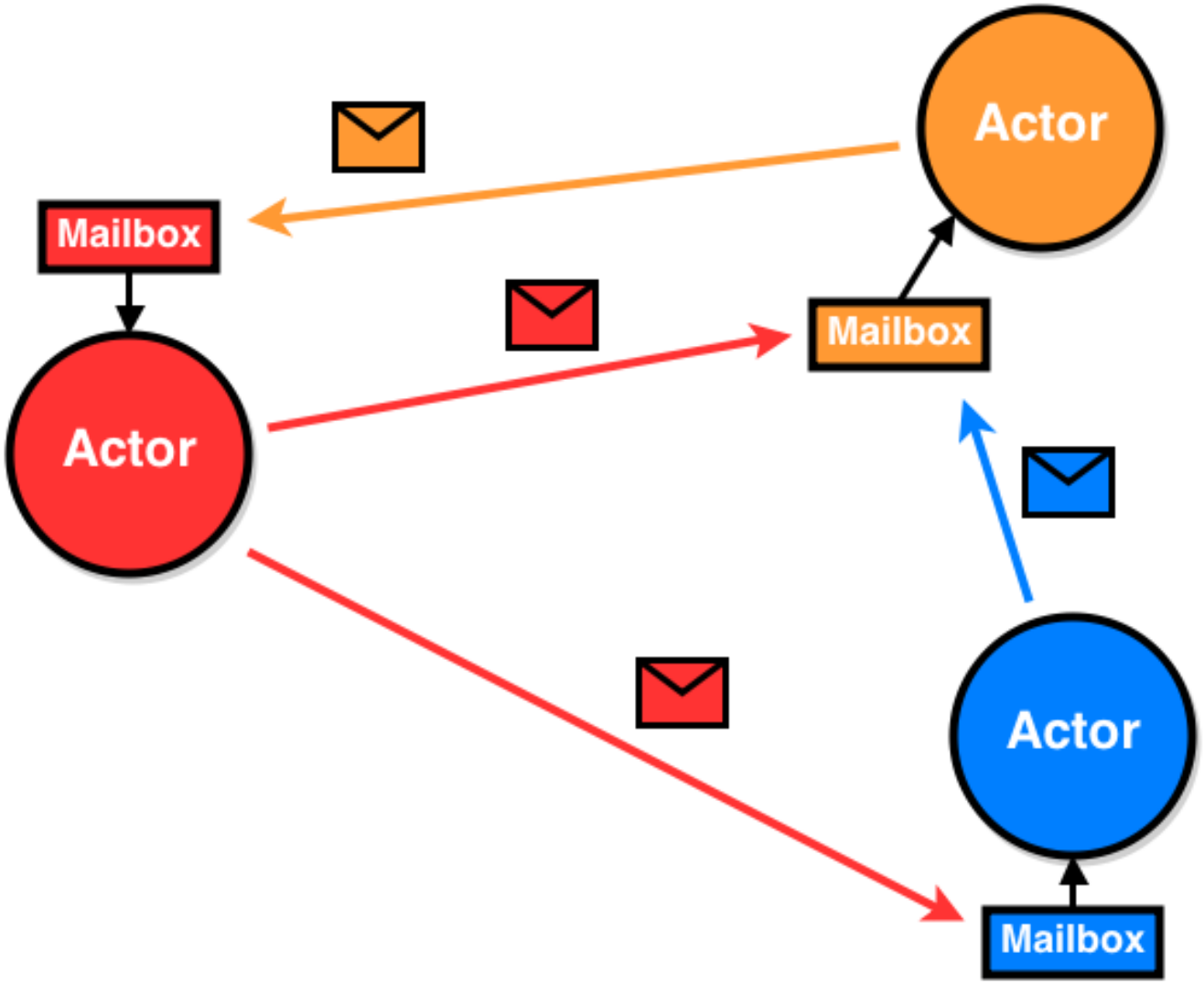
Clojure-way: очереди!

- а если использовать `M:N parallelism`?
  - запускаем N задач на M настоящих тредах (тредпул)
  - при блокировке задачи поток не блокируется, а берёт на выполнение следующую задачу

Actor system?

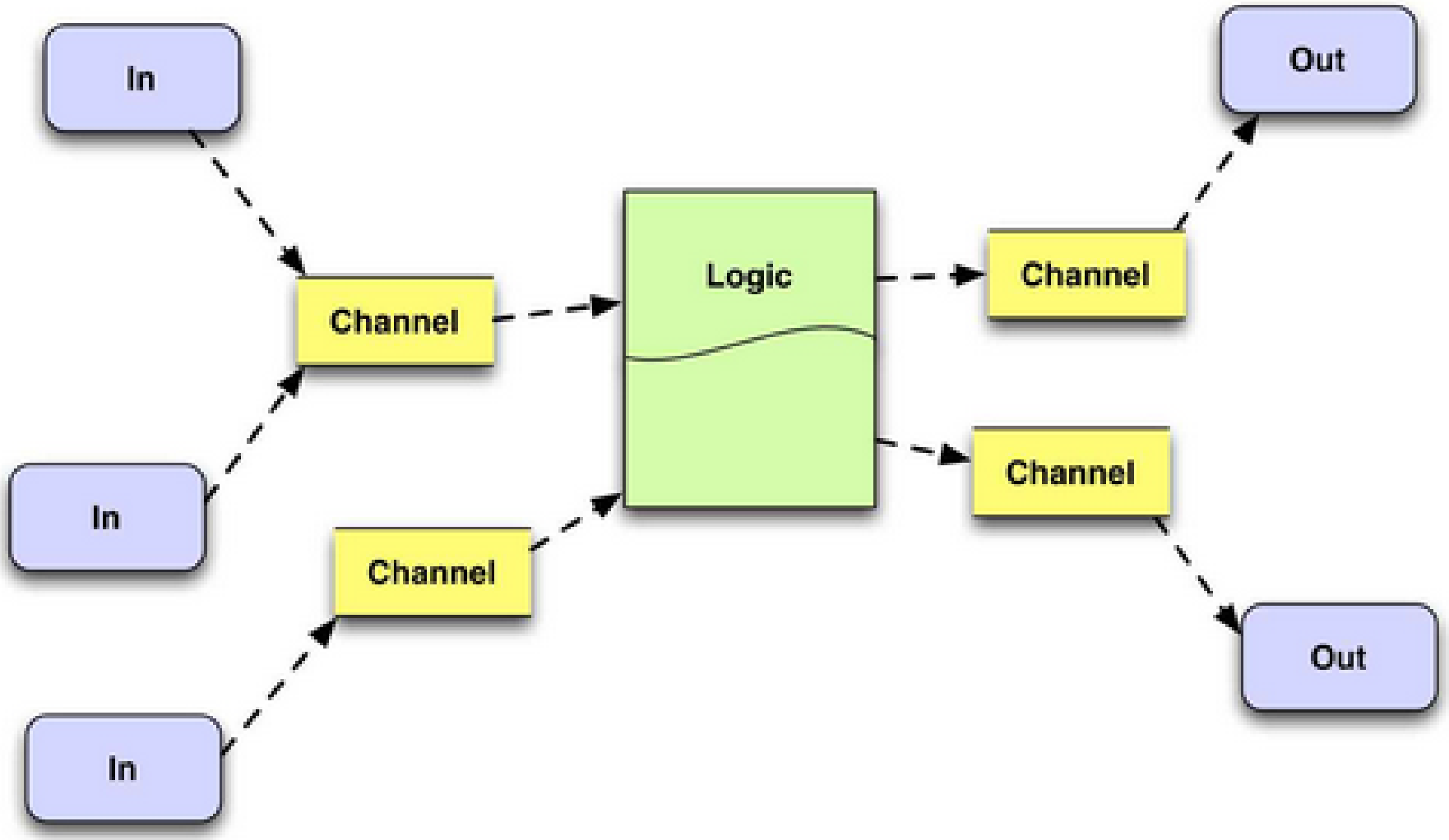
- отсутствует в Clojure **by-design**
- набор акторов, каждый (непрозрачно) хранит своё состояние
- работают конкурентно
- последовательное изменение состояния согласно **внутренним** правилам
- каждый актор характеризуется
  - почтовым ящиком для входящих сообщений
  - правилами реакции на входящие сообщения
- ? адресация по идентификаторам или передачей объектов
- ? иерархичность, супервайзеры, обработка ошибок

Actor system



Communicating sequential processes (CSP)

> Good programs should be made of processes and queues.



Communicating sequential processes (CSP)

- Erlang
- Limbo
- Go
- и многие другие современные языки (за счёт библиотек)

CSP

Данные "протекают" по конвееру, состоящему из

- каналов
  - транспорт между процессами
  - средство координации процессов
  - first class, можем передавать в и возвращать из функций и других каналов
- процессов
  - обработка данных с входных каналов и передача результатов на выходные
  - последовательная обработка сообщений
  - все процессы работают параллельно

Процессы

real thread -- `1:1 parallelism`

```
(thread <body>)
```

- код выполняется **на другом** треде
  - сразу после создания возвращает поток управления
- в случае блокировки, выполняющий **поток блокируется**
- возвращает канал, в который попадёт результат выполнения тела

Процессы

IOC thread -- `M:N parallelism`

```
(go <body>)
```

- мы пишем последовательный код, компилятор переписывает его на *magic callback hell* за нас (похоже на async/await)
  - сразу после создания возвращают поток управления
- в случае блокировки "паркуются"
  - после появления ожидаемого сообщения будет разбужен и продолжит работу
- возвращают канал, в который попадёт результат выполнения тела

Каналы

- FIFO очередь с операциями `put`, `take`
- **блокирующие** по умолчанию, но поддерживают буферизацию
  - средство координации
- multi reader / writer
- каналы с бесконечным буфером **отсутствуют**

Basic channel API

Создание каналов

```
(let [ch (chan)] <body>)      ;; unbuffered
(let [ch (chan 10)] <body>)   ;; fixed buff = 10
(let [ch (chan buf)] <body>) ;; special buffer
(timeout 10)                  ;; timeout channel, 10ms
```

Basic channel API: Создание каналов

тип буфера	семантика	пример
unbuffered	"рандеву"	`(chan)`
fixed	"рандеву" + сохранение	`(chan 10)`
sliding	выкидываем старые	`(chan (sliding-buffer 10))`
dropping	выкидываем новые	`(chan (dropping-buffer 10))`

Basic channel API

Примитивные операции:

	go (park)	thread (block)	external
put	`(>! ch val)`	`(>!! ch val)`	`(put! ch val)`
take	`(<! ch)`	`(<!! ch)`	`(take! ch fn)`
offer			`(offer! ch val)`
poll			`(poll! ch)`

Простые примеры

```
(def c (chan))
(future (>!! c 16))
(<!! c)          ;; => 16
(<!! c)          ;; block forever!
```

```
(def c (chan))
(>!! c 1)        ;; block forever!
```

```
(<!! (timeout 100))    ;; will block for 100 ms
```

Multiplexing/demultiplexing

```
(pipe from to)
(merge ch1 ch2 ch3 ...)   ;; => chan with all items from ch*
(split predicate ch true-ch false-ch)
```

```
(def mult-ch (mult ch))   ;; creates multiplexer
(tap mult-ch ch)
(untap mult-ch ch)
```

```
(def mix-ch (mix ch))     ;; creates mix of channels
(admix mix-ch ch)
(unmix mix-ch ch)
```

```
(pub ch topic-fn)
(sub pub-ch topic ch)
(unsub pub-ch topic ch)
```

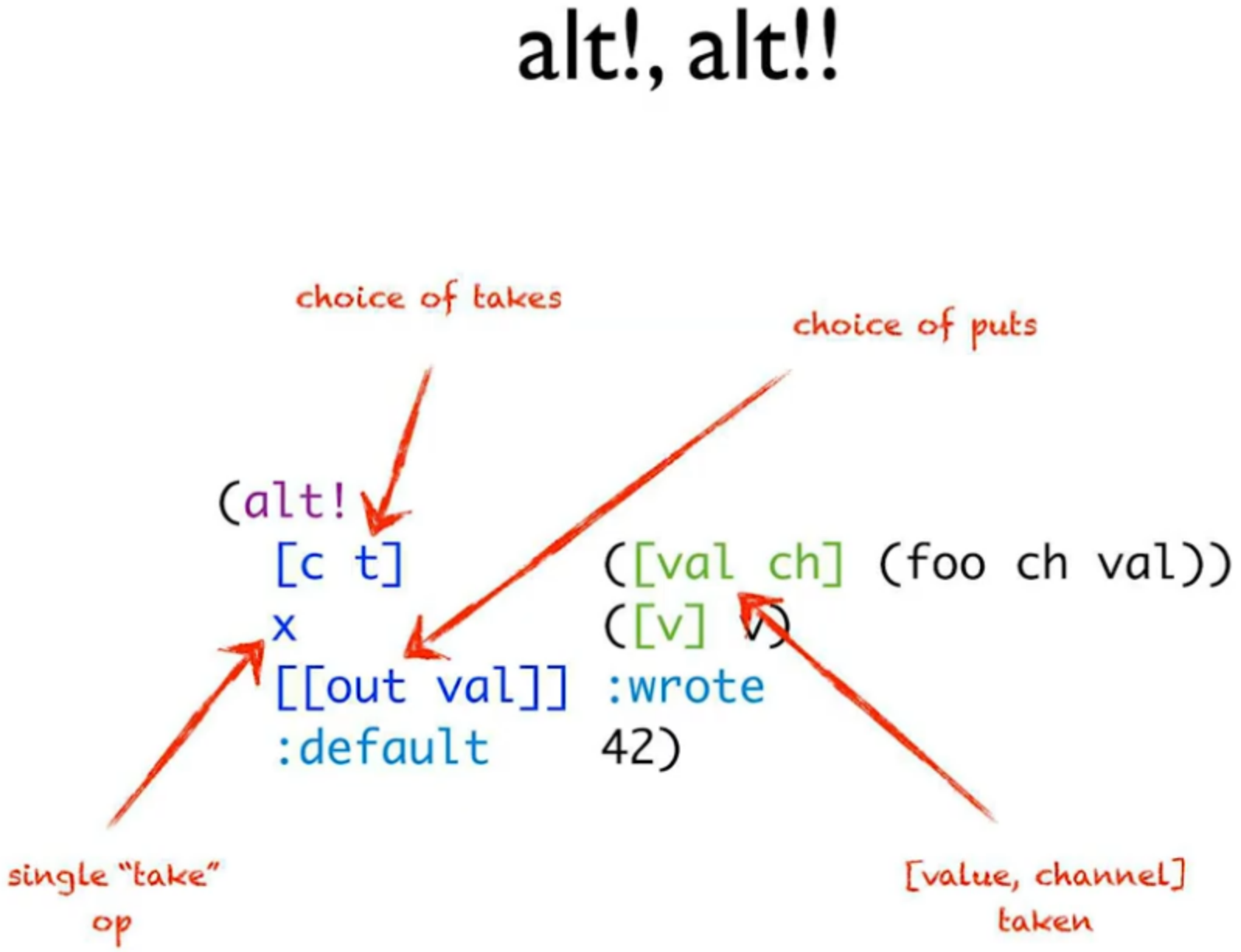
Селективные операции на каналах

```
(alt! & clauses)
```

```
(let [timeout-ch (timeout 1000)]
  (alt!
    timeout-ch :timed-out
    [[out-ch val]] :sent))
```

пытаемся отправить `val` в канал `out-ch`, пока не истёк таймаут

Селективные операции на каналах



Счётчик

```
(defn counter [in out]
  (go-loop [acc 0]
    (alt!
      in ([v] (recur (+ acc v)))
      [[out acc]] (recur acc))))
```

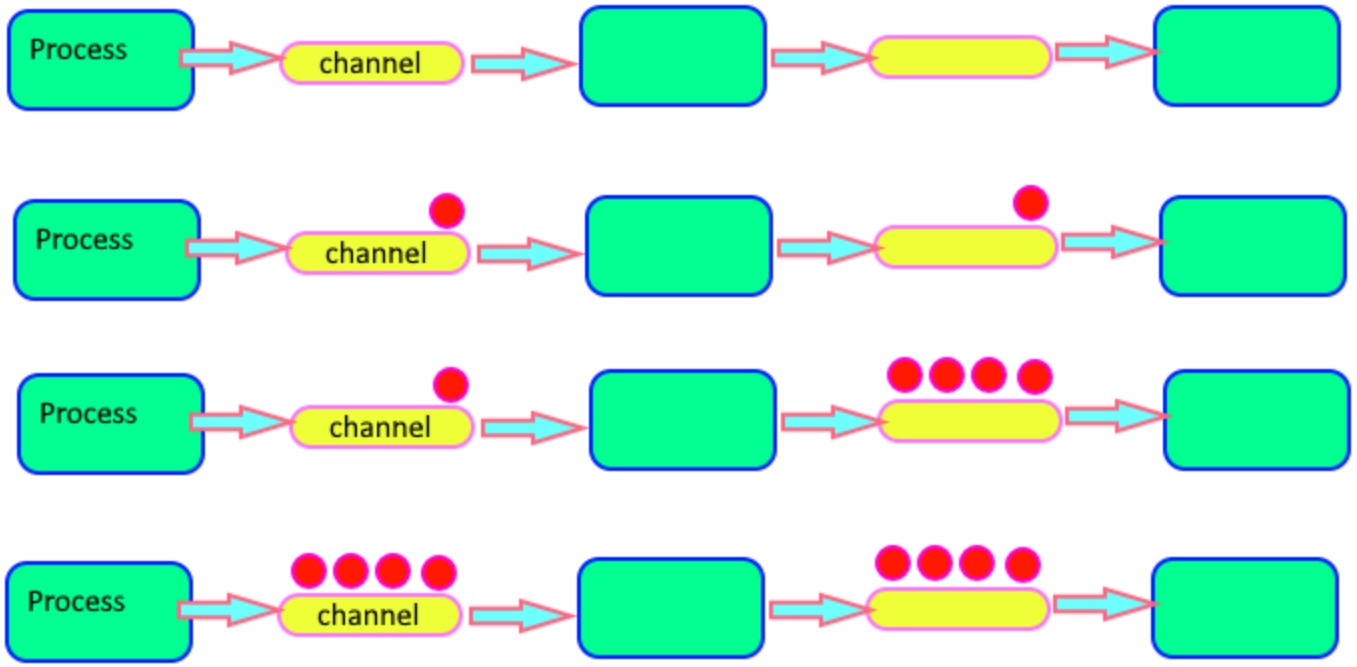
- если приходит значение `v` на канал `in`, то увеличиваем аккумулятор
- если на `out`-канале есть запрос, выдаём в него аккумулятор

[Demo]

Backpressure

- если данные приходят быстрее, чем их получается обработать, образуется затор
- (допустим) есть неограниченные размером каналы
  - чем больше проходит времени, тем больше сообщений копится
- лучше распространить замедление работы до начала конвеера, чем сломать систему
  - если дошли до самого начала -- 503 (Service Temporarily Unavailable)

Backpressure



Принципы построения приложений

что?	как?
вызовы и передача аргументов	посылка сообщений с входными данными
конструирование системы	передача каналов в компонены
обработка ошибок	компоненты не кидают ошибок, супервайзеры
состояние	функциональное или "unified succession model"

Спасибо за внимание!