

Разработка мобильных приложений

Лекция 5

Параллелизм: процессы, потоки и корутины

Ключев А.О. к.т.н., доцент ФПИиКТ Университета ИТМО

Санкт-Петербург

2025

Литература

- Лав, Роберт. Ядро Linux: описание процесса разработки, 3-е изд. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2013. — 496 с. :
- Примеры исходных текстов к лекции -
 - <https://github.com/kluchev/rmp-kotlin-lecture>
- [Эдвард Ли. Проблемы с потоками.](#)
- Youtube
 - [Андрей Бреслав — Что такое Kotlin? Введение](#)
 - [Лекция 1 | Kotlin: практика разработки | Андрей Бреслав | Лекториум](#)
 - [Андрей Бреслав — На плечах гигантов: языки, у которых учился Kotlin](#)
 - [Андрей Бреслав — Kotlin для Android: кратко и ясно](#)
 - [Роман Елизаров — Корутин в Kotlin](#)
 - [Корутин в Kotlin на сервере \(Роман Елизаров\)](#)
 - [Андрей Бреслав — Асинхронно, но понятно. Сопрограммы в Kotlin](#)

Разработки ЛИТМО, кафедры ВТ и лаборатория микропроцессорной техники (XX век)

- Компьютер ЛИТМО-1
- Компьютер ЛИТМО-2
- Разработка компиляторов для программирования встроенных систем.
- Разработка микропроцессора (конец 80-х, начало 90-х).
- Разработка персональных компьютеров серии Дельта на базе K580ИК80, Z80 (CP/M), K1810BM86 (DOS). Системное ПО написано с использованием своих инструментальных средств (компилятор и язык программирования).
- Разработка учебных стендов «Микроконструктор» на базе K580ИК80
- Разработка видеотерминалов VT100 для многотерминальных комплексов на базе Unix/Xenix и PC MOS
- К сожалению, все эти разработки не выдержали конкуренции после развала СССР =(

Бэкграунд, мои основные разработки или почему мне повезло

- Системы управления освещением (начиная с середины 90-х до настоящего времени), управление и сбор информации с большого количества контроллеров.
- Высокоуровневые ассемблеры для Intel 8080 и 8086 на базе FORTH
- Инструментальные средства со скриптовым языке на базе FORTH (терминал, сборка загрузочных модулей (аналог makefile), загрузка в различные контроллеры (поддержка PM3P, JTAG и др., аналог OpenOCD), отладка и тестирование.
- Простая RTOS (Intel 196)
- Загрузчики для различных контроллеров
- Разработка аппаратуры (различные контроллеры и модули ввода-вывода)
- Виртуальная машина для ПЛК SVM v1.0 (Stack Virtual Machine) для PIC16 с маленьким футпринтом и размещением IR в E2PROM. Целевой компилятор FORTH для SVM v1.0
- Контроллерные сети, вагонные и аэродромные кондиционеры, измерители влажности, приборные контроллеры, контроллер для управления дизельгенераторами,...
- Виртуальная машина для ПЛК SVM v3.0 со специфическим интеропом с библиотеками на Си. Компилятор языка программирования STL (Structured Text Lite) для ПЛК на базе IEC-61131-3, фронтэнд компилятора на базе Parser Generator (Lex/Yacc), генерирующий IR для SVM-3.0
- Разработки под Android (умный дом, система освещения и т.п.) на Java
- Различное серверное ПО (C/C++/C#/Python/Java/Kotlin)
- Различные программы с GUI под Windows на C# и VB.NET
- Генераторы сайтов со статическим HTML
- Проекты на JavaScript (Angular, Node.js)
- Система ITMO cLAB
- Система LabScript (мультипарадигмальный язык программирования и распределенные гетерогенные виртуальные машины)

Специфика разработок

- Весь спектр вычислительной техники:
 - Разработка инструментальных средств (ассемблеров, компиляторов, линкеров, сборщиков, загрузчиков, виртуальных машин,...)
 - Разработка аналоговой и цифровой аппаратуры (в том числе на FPGA)
 - Разработка ПО для встроенных систем:
 - Разработка ПО низкого уровня (драйверы, ядро RTOS,...)
 - Разработка прикладного ПО реального времени
 - Разработка серверного ПО, в том числе высоконагруженного
 - Разработка прикладного ПО с интерфейсом пользователя (нативные и Web реализации).
- Другими словами, деятельность не сводится к CRUD

Что я из этого всего вынес? Это мое личное, хулиганское, неформальное суждение.

- Хорошие и полезные книги это боль. На следующем слайде я объясню почему.
- Архитектура системы очень важна, но непонятно как ее правильно придумать, как реализовать и как передать свои мысли другим. А еще я не знаю после 30 лет практики преподавания, как научить вас и не знаю того, у кого это можно спросить. Время показало, что ничего удобоваримого пока еще не придумали. Архитектура это боль.
- Параллелизм очень важен, но непонятно, как его реализовывать. Книг нет. Параллелизм это тоже боль, причем университеты готовят скорее каких-то религиозных фанатиков для какой-то ужасной секты, вооруженных фреймворками, новыми языками, работающими на старых принципах, потоками и семафорами, а не специалистов.
- Понятие модели вычисления гениально в своей простоте и значимости, но почему его никто не знает/не понимает/не изучает (спойлер – ответ на следующем слайде)? Возможно это надо выстрадать. Я выстрадал в куче проектов.
- Языки программирования не особо адекватны решаемым задачам. Я их перепробовал много, больше всего понравился Perl, но только для обработки текстов, все остальное – отстой. Даже Kotlin и Go из той же оперы. Я бы сказал, что Go и Kotlin это тоже доведенное до совершенства «нечто на костылях», хотя они лучшие, из того что я пробовал использовать в коммерческих проектах.
- Операционных систем реального времени не бывает. RTOS и Дед Мороз это сказки, увы.
- Вычислительная техника развивается эволюционно, а не осмысленно.
- Человечество гораздо глупее, чем мы все думаем.
- Программисты прекрасно умеют занимать весь объем ресурсов (почему – ответ на следующем слайде), которые им предоставили аппаратчики. И таки да, программа это газ.

Немного про IQ

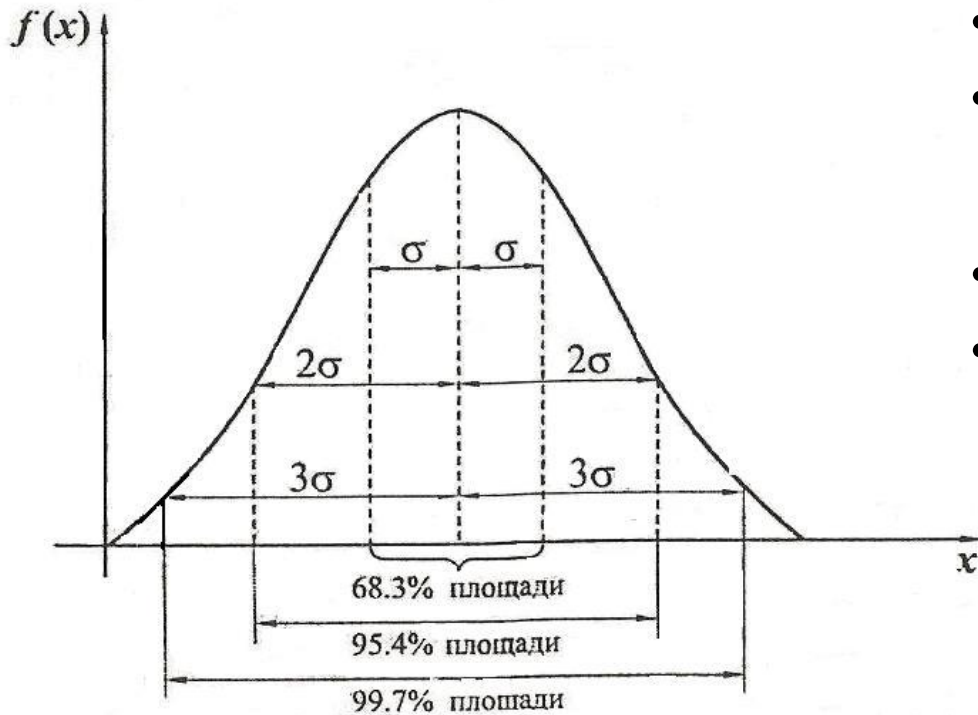


Рис. 2.3. Нормальный закон распределения

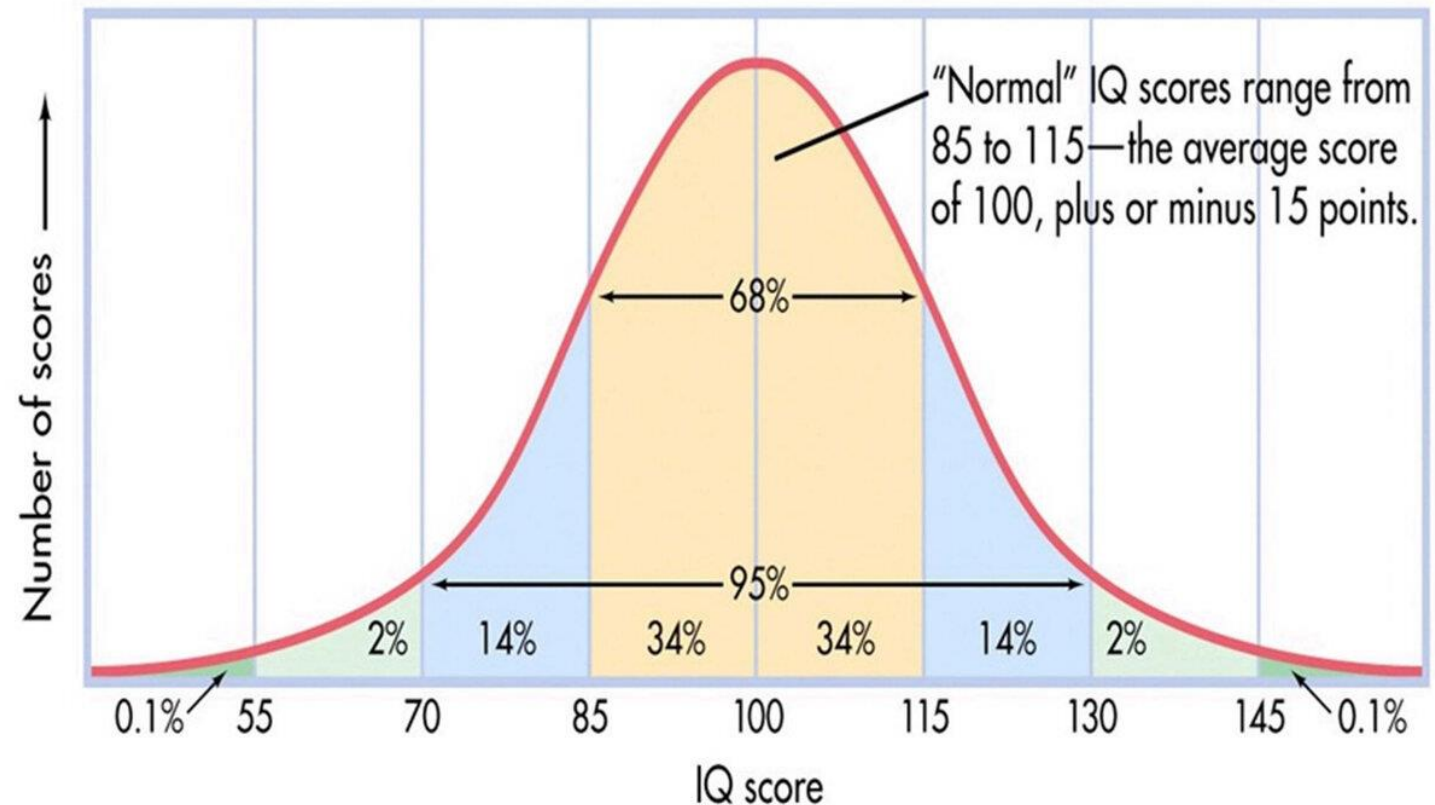
- IQ - тест на уровень интеллекта
- Тесты IQ разработаны так, чтобы их результаты можно было оценить с помощью шкалы нормального распределения.
- См. [Очередная статья об IQ](#)
- Мое личное мнение – тест на IQ слишком упрощен, так как он должен подходить буквально для всех и не учитывает многие вещи. Например, тест IQ может показать, что джуниор-программист умнее сеньора, но джуниор не может решать сложные задачи, а сеньор может. У разных людей разная разная степень тренировки мозга (например, увеличение результативности прохождения теста на IQ достигается тренировкой в прохождении подобных тестов).
- Несмотря на все недостатки, тест на IQ грубо показывает характер интеллекта крупных масс людей на планете.

Как все это относится к нашей теме?

- Большинство рабочих мест для средней части графика (иначе где вы возьмете сотрудников?)
- Технологии слабо меняются по своей глубинной сути (они меняются только во внешней форме) так как дорого переучивать сотрудников.
- Книги и журналы – для средней части графика (иначе, кто все это купит?)
- Фильмы, новости, развлечения,...

Ну, в общем вы поняли.

Возможно нас спасет ИИ, но это не точно.



Проблема с литературой и документацией



- Проблемы умных текстов (немного утрируя):
 - Умные тексты мало кто может написать (умных людей мало);
 - Умные тексты трудоемко писать (у умных людей много другой, интересной и важной работы);
 - Умные тексты мало кто может понять;
 - Умные тексты никто не хочет издавать, так как покупателей мало.
- Хороший источник умных текстов – научные статьи, но они как правило не объясняют основ, а показывают решение крайне узких задач, довольно далеких от реальной жизни. Кроме того количество статей огромно, а доступ к ним сильно ограничен. К сожалению часто бывает, что научная статья это просто формальное средство отчетности по очередному гранту, не содержащая для читателя ничего полезного. См. книгу «Уродливая вселенная», по мнению автора уже 50 лет нет значительных открытий в теоретической физике.
- Основы вычислительной техники размазаны ровным слоем по огромному количеству научных трудов в области компьютерных наук, электронике, информатике, разным разделам математики и т.п.
- В книгах для «широкого потребления» сложные темы обходят стороной, а изложение материала сделано в виде большого количества примеров (как в stackoverflow). Очень популярны книги в стиле сборника рецептов.
- В подавляющей массе литературы объясняется как сделать что-то, но не объясняется почему и какая концепция лежит в основе.

Контекст

В данной лекции речь в основном идет о ядре Linux, так как это составная часть Android



Рост сложности технологий

- До 2010 этой темой никто кроме узких специалистов работающих в сфере науки особо не озадачивался, в основном решались относительно простые задачи. Большое количество параллельных процессов нужно было в основном в различных научных расчётах, в моделировании и т.п.
- Начиная примерно с 2010 года начали появляться массовые мобильные устройства (компактные ноутбуки, планшеты, смартфоны, головные устройства автомобилей, терминалы и т.п.), постоянно подключенные к сети Интернет.
 - Появились массовые, доступные Интернет-сети на базе беспроводных технологий (GSM, Wi-Fi,...) с широким покрытием.
 - Появился Интернет вещей (IoT), беспроводные сенсорные сети, умные дома, умные города, интеллектуальные транспортные системы и транспортные средства с автопилотом.
 - Массово стали появляться новые сервисы (интернет-банкинг, онлайн покупки, онлайн игры, развлекательные сайты, стриминговые сервисы (видео и звук) и т.д. и т.п.)
 - На порядки выросла нагрузка на серверное оборудование
 - Начался бурный расцвет облачных технологий, начали развиваться микросервисные технологии, брокеры сообщений.
 - Производительность мобильных устройств начала быстро расти
 - Начали активно развиваться инструментальные средства для асинхронного программирования.
- Другими словами – круг решаемых задач существенно расширился, а сами задачи кратно усложнились.

Зачем нужен параллелизм?

- Параллелизм позволяет решать несколько задач одновременно, разделяя процессорное время одного или нескольких процессорных ядер между большим количеством задач, например, решать 100 задач на четырех процессорных ядрах одновременно так, как будто у нас на самом деле 100 отдельных CPU.
- Распараллеливание вычислений очень часто позволяет упростить алгоритм (особенно это актуально в управляющих системах).
- В мобильных системах в настоящий момент решаются более сложные задачи, чем решались на настольных компьютерах 10-15 летней давности (современные игры, мобильная фотография, использующая слабую оптику и сложную обработку фото на базе нейросетей (это практически уничтожило продажи дорогих зеркальных фотоаппаратов), сложные сетевые сервисы, такие как навигация и т.п.)
 - Решение сложных задач на мобильных платформах становится в принципе возможным.
 - Приложение можно написать таким образом, чтобы сложные процессы решаемые «под капотом» не влияли на плавность работы интерфейса пользователя.

Механизмы параллелизма (в контексте Android)

- Средства реализации параллелизма
 - Процессы [ядро Linux]
 - Поток [ядро Linux, JVM]
 - Корутин (сопрограммы) [Компилятор Kotlin]
- Средства межпроцессного взаимодействия (IPC, Inter-process communication)
 - Каналы, очереди сообщений [ядро Linux]
 - Binder [ядро Linux]
 - Общая память [ядро Linux]
 - Потокзащищенные каналы [Библиотеки Java/Kotlin]
 - Intent [Библиотеки Android]
 - Channel [Библиотеки Kotlin для корутин]
 - Атомарные переменные [Библиотеки Java и Kotlin]
- Средства синхронизации
 - Семафоры и мьютексы
- Средства управления процессами и потоками
 - create/close, start/stop

Виды многозадачности в рамках одного CPU

- Вытесняющая или приоритетная (preemptive) многозадачность
 - Один процесс прерывает или вытесняет другой при помощи переключателя задач. Защититься от переключателя задач можно с помощью критических секций (это напрямую практикуется в Windows API и косвенно, через мьютексы в POSIX (Linux, MacOS X и т.д.)) или путем выключения переключателя задач (такое можно встретить только в самых простых операционных системах, например в FreeRTOS).
- Согласующая или кооперативная (cooperative) многозадачность
 - Один процесс передает управление другому по собственному желанию. Если процесс не передаст управления сам, то этот процесс будет выполняться, а остальные процессы будут простаивать.

В чем суть и как реализуется параллелизм в ОС?

- Что такое параллелизм на самом деле?
 - Это реализация модели вычислений потоков данных, например сетей процессов Кана (Kahn Process Network, PN) на базе однопроцессорной или многопроцессорной вычислительной аппаратуры, работающей на базе модели Фон-Неймана
- Как реализуется?
 - Переключатель задач – реализует модель вычислений PN, прерывая задачи и передавая управление. Каждая задача выполняется кусочками. Из-за этого переключателя задач и возникают гонки, так как задачу могут «разрезать» в тот момент, когда данные еще не до конца обработаны.
 - Планировщик – выбирает из списка всех задач самую важную и готовую к исполнению задачу.

Переключатель задач

- Переключатель задач – программа активирующая задачи готовые к исполнению (т.е. содержащиеся в очереди задач).
- Переключатель выполняет следующие действия:
 - Сохраняет контекст текущей задачи;
 - Выбирает из очереди новый дескриптор задачи;
 - Восстанавливает контекст выбранной задачи.

Контекст процесса (в общем виде)

- Контекст процесса это набор рабочих переменных определяющих текущее состояние.
- В контекст процесса входят:
 - IP (Instruction Pointer) указатель на следующую инструкцию процессора.
 - SP (Stack Pointer), указатель стека.
 - Различные флаги ЦП
 - Используемые процессом регистры общего назначения

Очередь задач

- Очередь задач содержит множество дескрипторов задач готовых к исполнению. Порядок постановки задач в очередь определяется планировщиком. Переключатель задач вынимает дескрипторы из очереди.
- Дескриптор задачи – некое число, однозначно идентифицирующее задачу. Обычно это индекс в таблице содержащей информацию о задачах.

Алгоритм планирования

- Алгоритм планирования это набор правил, в соответствии с которыми осуществляется выбор задачи (процесса) для обслуживания.
- Синонимы алгоритма планирования: метод, стратегия, политика планирования.

Планировщик

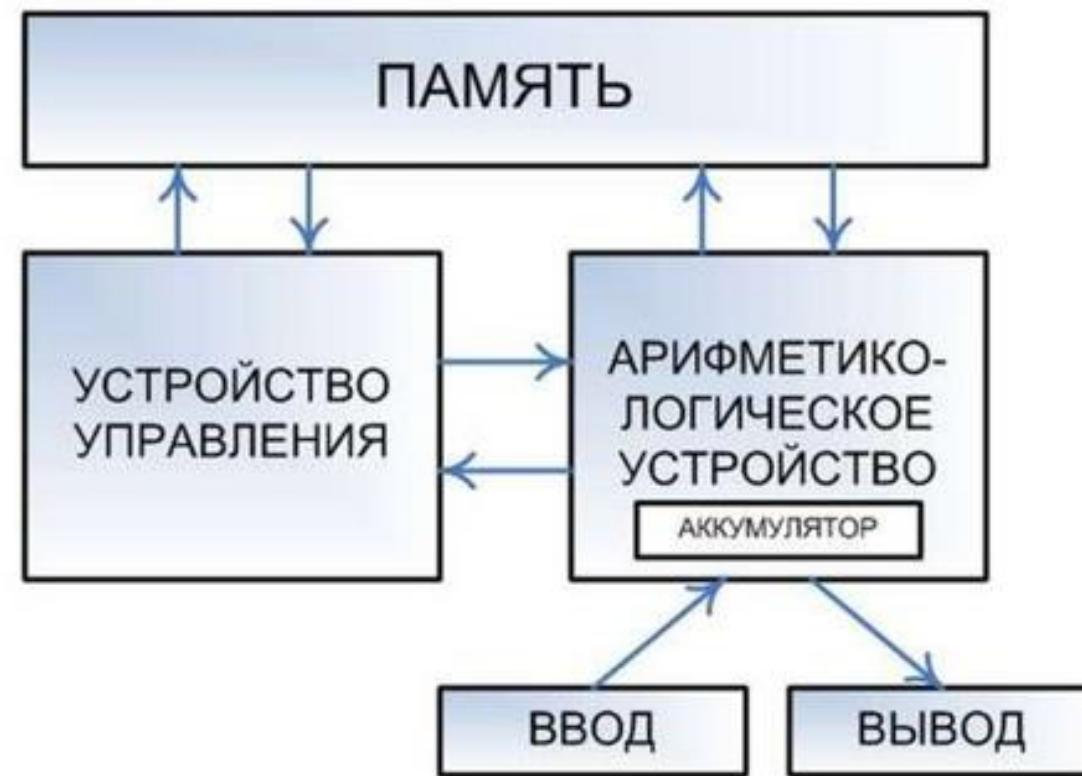
- Планировщик задач – это программа, реализующая алгоритм планирования процессов.
- Функции планировщика:
 - выбор очередной задачи на выполнение;
 - слежение за крайними сроками задач;
 - активация задач.

Приоритет

- Приоритет – численное значение, приписываемое задаче операционной системой и отражающее ее важность.
- Чем выше приоритет, тем больше процессорного времени отдается задаче и тем быстрее она выполняется.

Машина Фон-Неймана, основные принципы

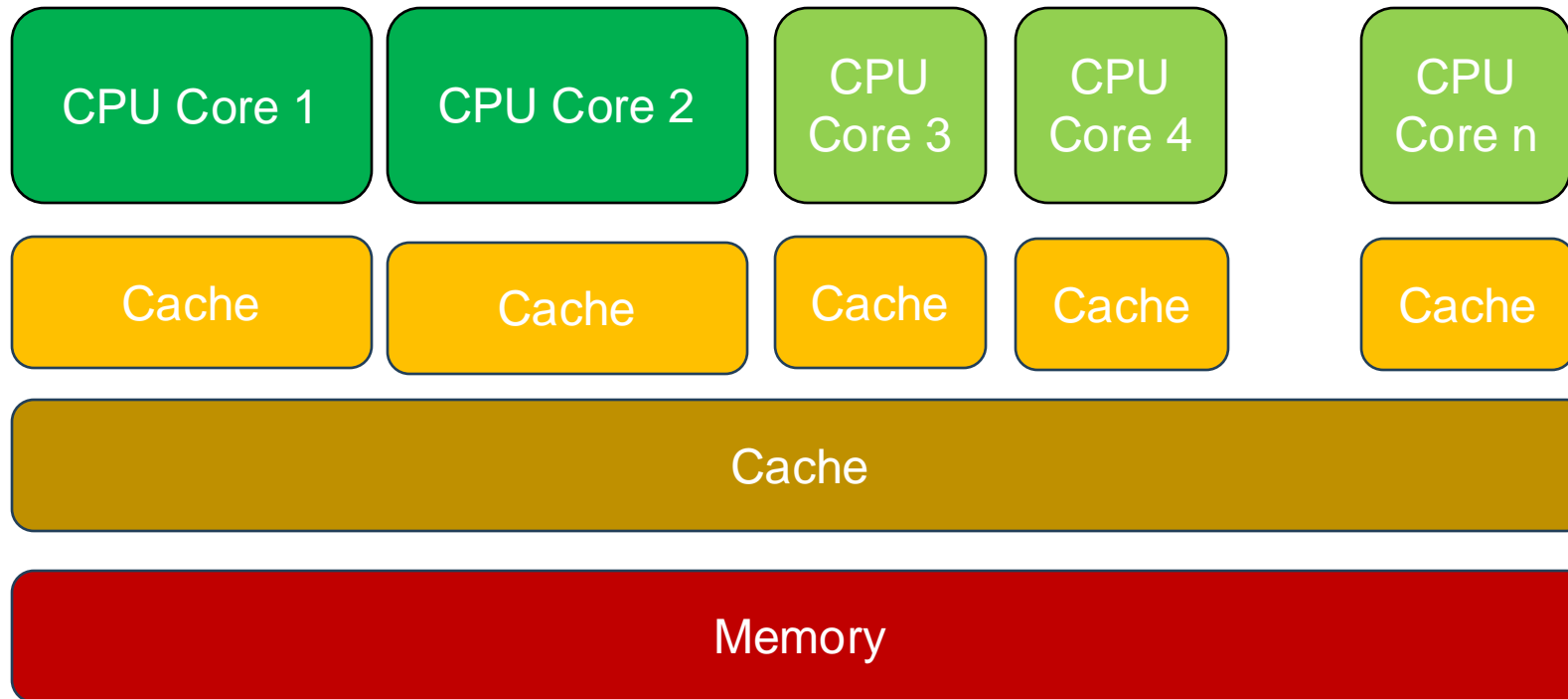
- Основные принципы:
 - **Использование двоичной системы счисления.**
 - **Программное управление.** Работа компьютера контролируется программой, состоящей из набора команд. **Команды выполняются последовательно друг за другом.** Созданием машины с хранимой в памяти программой было положено начало тому, что мы сегодня называем программированием.
 - **Память компьютера используется не только для хранения данных, но и программ.**
 - **Ячейки памяти компьютера имеют адреса, которые последовательно пронумерованы.**
 - **Существует возможность условного и безусловного перехода в процессе выполнения программы.**



Современные процессоры сильно отличаются от машины Фон-Неймана

- Прерывания, которые позволяют осуществить метасистемный переход (см. Турчин В. Ф. Феномен науки. Кибернетический подход к эволюции) от машины Фон-Неймана к Process Network
- Прямой доступ к памяти (ПДП, DMA)
- Контроллеры и процессоры ввода-вывода
- Активные ячейки памяти (в которых не просто хранятся данные, а туда «замаплены» регистры данных и управления различных устройств ввода-вывода).
- Аппаратный параллелизм на разных уровнях, от конвейеров и АЛУ, до многоядерной архитектуры.

Упрощенная архитектура современной вычислительной системы

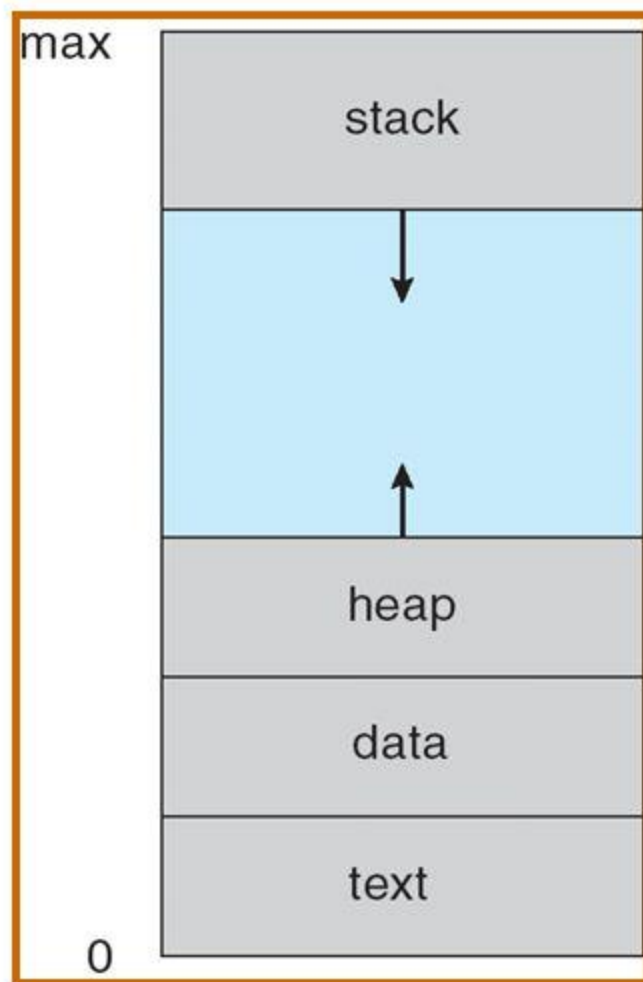


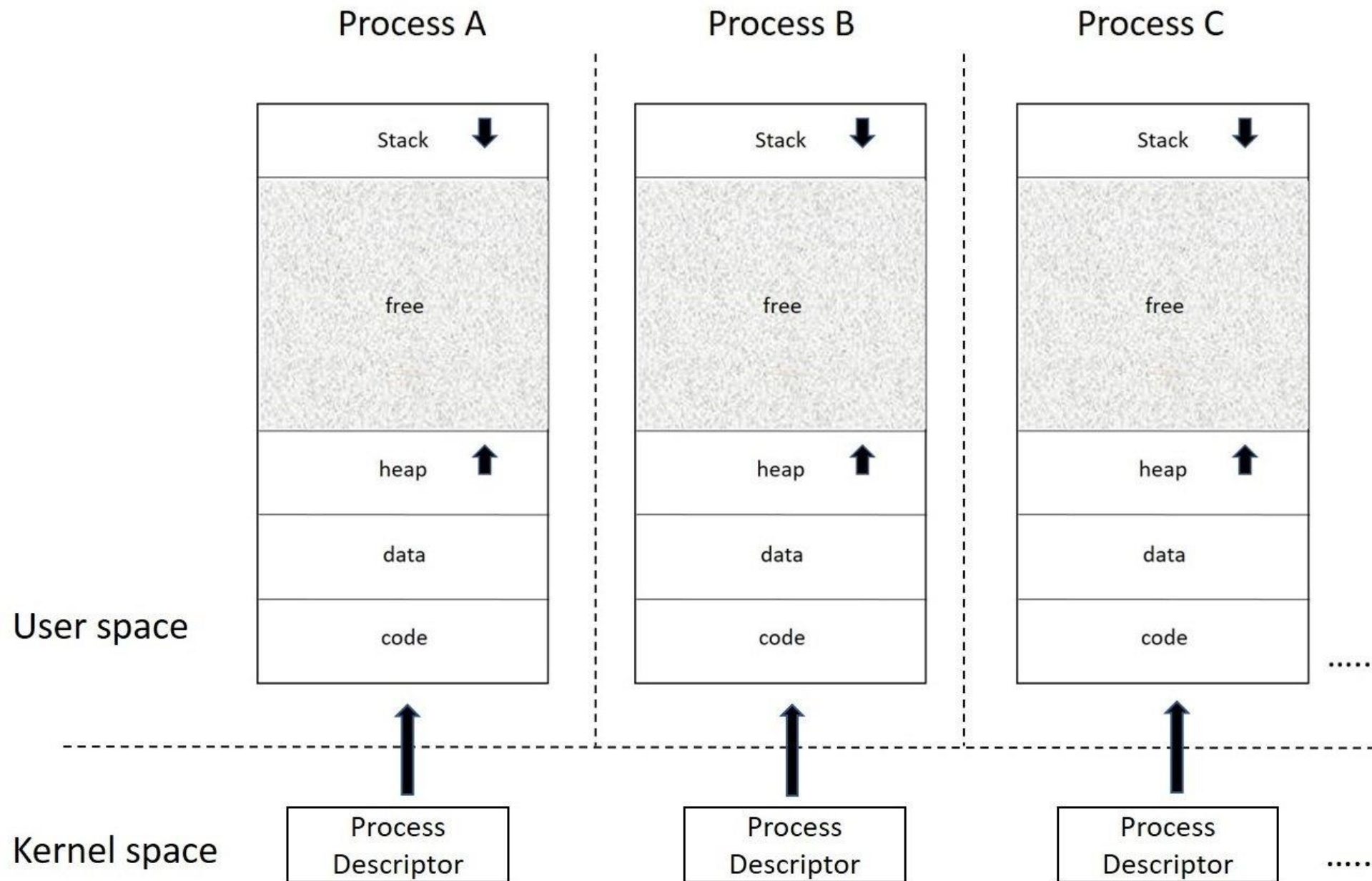
Управление процессами в ядре Linux

- Планировщик Linux выполняет процедуру планирования процессов за фиксированное время
- CFS (Completely Fair Scheduler) - полностью справедливый планировщик см. <https://docs.kernel.org/6.1/scheduler/sched-design-CFS.html>
 - CFS для программиста выглядит как один виртуальный CPU (работающий на реальной многоядерной аппаратной платформе), выделяющий каждой задаче одинаковые части своего процессорного времени (например, для двух задач это будет 50%, для трех 33%, для четырех 25% и так далее).

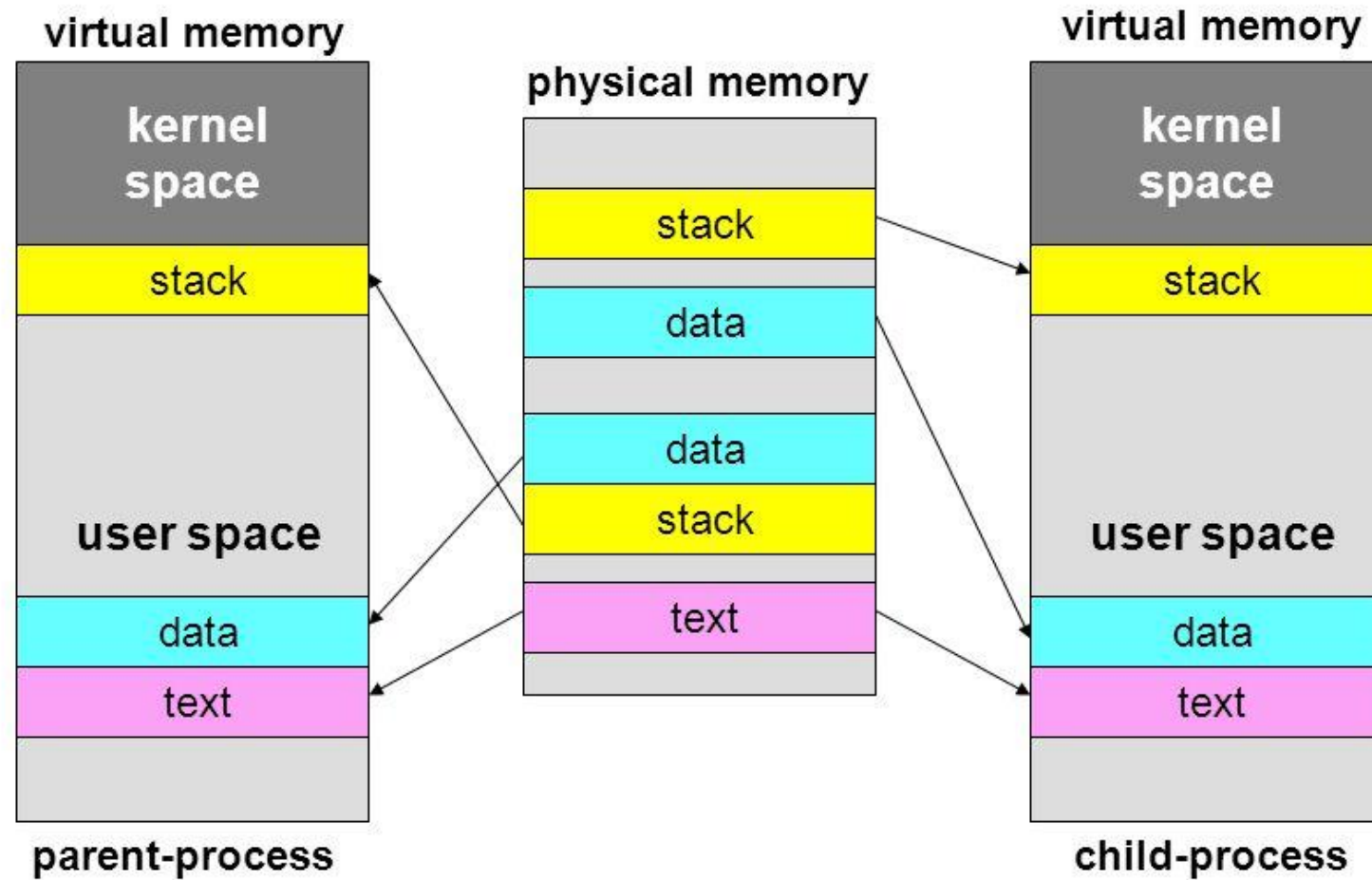
A Process In Memory

- Stack:
 - Local variables declared inside functions
 - Function Parameters
- Heap:
 - Dynamically allocated memory
- Data:
 - Global variables, etc.
- Text:
 - Program Instructions





Similar memory-mappings



Процесс

- Процессы в Linux работают изолированно, они не знают о существовании друг друга, о большой физической памяти и нескольких процессорных ядрах. Каждый процесс живет в своем виртуальном мире, в котором есть один единственный виртуальный процессор и собственная память, которая не видна другим процессам.
 - За выделение процессорного времени отвечает планировщик.
 - За выделение физической памяти – механизм виртуальной памяти.

fork

- `fork()` (вилка) - системный вызов в Unix-подобных операционных системах, создающий новый процесс (потомок), который является практически полной копией процесса-родителя, выполняющего этот вызов.
- При вызове `fork()` возникают два полностью идентичных процесса. Весь код после `fork()` выполняется дважды, как в процессе-потомке, так и в процессе-родителе.
- См. https://www.opennet.ru/docs/RUS/linux_parallel/node7.html

Пример использования fork()

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
    pid_t pid;
    switch(pid=fork()) {
        case -1: printf("ERROR!\n"); break;
        case 0:  printf("CHILD:  %d\n", pid); break;
        default: printf("PARENT: %d\n", pid); break;
    }
    return 0;
}
```

```
$ gcc test_fork.c
$ ./a.out
PARENT: 50352
CHILD:  0
```

Приоритет процесса и квант времени

- В Linux приоритет процесса определяется параметром **nice** который может принимать значения от -20 до 19. -20 высший приоритет, 19 – низший. Значение по умолчанию – 0.
- Каждому уровню приоритета (nice) соответствует свой вес (**weight**), значение которого определяет время, выдаваемое процессу для работы. Чем больше приоритет, тем больше времени выделяется.

nice: см. команды top и ps

```
top - 11:07:27 up 3 days, 1:52, 2 users, load average: 0,00, 0,02, 0,00
Tasks: 368 total, 1 running, 367 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0,0 us, 0,1 sy, 0,0 ni, 99,8 id, 0,1 wa, 0,0 hi, 0,0 si, 0,0 st
МИБ Mem : 32029,9 total, 22477,1 free, 6117,5 used, 3435,3 buff/cache
МИБ Swap: 2048,0 total, 2048,0 free, 0,0 used. 25312,3 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1687	clickho+	20	0	47,8g	1,1g	370404	S	1,7	3,5	124:35.41	clickhouse-serv
1617	nx	20	0	1794480	113344	13672	S	0,3	0,3	4:25.98	nxserver.bin
1	root	20	0	166604	11992	8408	S	0,0	0,0	0:06.86	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.02	kthreadd
3	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	rcu_par_gp
5	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	slub_flushwq
6	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	netns

```
ps -eo pid,ppid,ni,comm
PID      PPID    NI  COMMAND
1         0      0  systemd
2         0      0  kthreadd
3         2    -20  rcu_gp
4         2    -20  rcu_par_gp
```

```
ps -elf
F S UID          PID      PPID    C  PRI   NI  ADDR  SZ  WCHAN    STIME  TTY          TIME CMD
4 S root           1         0    0   80    0  - 41651  -          окт27 ?          00:00:06 /sbin/init splash
1 S root           2         0    0   80    0  -      0  -          окт27 ?          00:00:00 [kthreadd]
1 I root           3         2    0   60  -20  -      0  -          окт27 ?          00:00:00 [rcu_gp]
```

Соотношение nice и weight

```
static const int prio_to_weight[40] = {  
    /* -20 */      88761,      71755,      56483,      46273,      36291,  
    /* -15 */      29154,      23254,      18705,      14949,      11916,  
    /* -10 */       9548,       7620,       6100,       4904,       3906,  
    /*  -5 */       3121,       2501,       1991,       1586,       1277,  
    /*   0 */      1024,        820,        655,        526,        423,  
    /*   5 */       335,        272,        215,        172,        137,  
    /*  10 */       110,         87,         70,         56,         45,  
    /*  15 */        36,         29,         23,         18,         15,  
};
```

Что такое поток?

- По идее – поток это легковесный процесс, имеющий общее адресное пространство с другими потоками и требующий меньше времени на переключение в ядре операционной системы.
- В Linux реализация потоков и процессов идентична, за исключением того, что процессы имеют изолированное адресное пространство, а потоки – совмещенное. Системные вызовы **fork** и **pthread_create** используют один и тот же системный вызов **clone**, с различным уровнем общего доступа (см. `man 2 clone`)

Поток vs процесс

- У потоков запущенных в рамках одного процесса общее адресное пространство
- У процессов свое собственное пространство. Один процесс не может записать данные в адресное пространство другого процесса просто так, для этого необходимо использовать механизм общей памяти, входящий в операционную систему.
- Поток – это легковесный процесс, то есть для обслуживания потока операционная система обычно тратит меньше ресурсов (ЦП и память).

Создание потока в Linux

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

void* test_thread(void *arg)
{
    int i;
    for( i = 0; i < 3; i++) {
        printf("hello world!\n");
        sleep(1);
    }
    return NULL;
}

int main(void)
{
    pthread_t t_id;
    int err;

    err = pthread_create(&t_id, NULL, &test_thread, NULL);
    if (err != 0)
        printf("Error :[%s]\n", strerror(err));
    sleep(5);
    return 0;
}
```

Потоки в JVM (Hot Spot) и ART

- Обе виртуальные машины используют NPTL (Native POSIX Thread Library) (если конечно HotSpot запустили поверх Linux =)
- Процесс виртуальной машины (HotSpot или ART) запускает потоки, которые мы создаем в Java/Kotlin
- Количество потоков запущенных в одном процессе (приложении Android) ограничено ядром Linux и может достигать нескольких тысяч штук.
 - `cat /proc/sys/kernel/threads-max`
 - 255197

Эдвард Ли, проблемы с потоками

Эдвард Ли в ИТМО 2019 г.



Проблемы потоков

- Недетерминированное поведение это такое поведение которое невозможно предсказать.
- Недетерминированное поведение программы приводит к неожиданному результату. Причинами недетерминированного поведения программы использующей потоки могут послужить:
 - Банальные ошибки в построении алгоритма, ошибки связанные с использованием опасных языков программирования типа C/C++, ошибки компилятора и аппаратуры (но об этом мы сейчас говорить не будем);
 - Проблемы связанные с параллелизмом:
 - Гонки
 - Блокировки

Проблема языков C/C++/Java/Kotlin, Go,...

- Эти языки построены на базе языков, придуманных в середине прошлого века для обычной машины Фон-Неймана, в которой не было даже намека на параллелизм. Эти языки не позволяют строить детерминированные программы с использованием:
 - Параллельных вычислений;
 - Учета времени выполнения.
- Если не верите, попробуйте на вышеозначенных языках построить программу, которая будет содержать в себе десятки, сотни, тысячи параллельных процессов и чем-то управлять в реальном масштабе времени. Желательно, чтобы после того как вы это напишите, автор программы сам встал «под мостом» во время испытаний. Не хотите? И правильно делаете... Скорее всего вы с этих испытаний живым не вернетесь.
- Новые «костыли», которые добавлены в эти языки для реализации параллелизма вызывают противоречия и сложности, приводящие к крайне тяжелым последствиям в программировании.
- Заменить языки на новые проблематично в силу огромной стоимости этого процесса, инерции индустрии и системы образования, которая выпускает каждый год сотни тысяч «средних» программистов для классических языков и приучены к старым «костылям». Никто не хочет увеличивать капекс индустрии за свой счет (это как с уплотнительной застройкой, проще воткнуть новый дом между двумя старыми, чем строить новый и удобный микрорайон и тянуть к нему дороги, канализацию, отопление, электричество и воду, а также строить школы, детские сады, магазины и поликлиники).

О костылях в ПО



▲ +84 ▼ 🐼 Vlad528 3 года назад

Просто у него плотность как у бетона. Нужно воде присвоить плотность ртути.

ответить

▲ +56 ▼ 🦉 Zmiillo 3 года назад

А если гусю кто-то в каком-нибудь костыле переприсвоит плотность урана?

ответить

▲ +103 ▼ 🐙 qwertum 3 года назад

Подключаем библиотеку воздушных шариков и прикручиваем к гусю. Затем алгоритмом высчитываем сколько шариков необходимо, чтобы гусь держался на поверхности, но при этом не улетал из-за шариков.

ответить

▲ +102 ▼ 🦉 Zmiillo 3 года назад

Допустим ты решил проблему, а потом ушел из проекта.

Пришел новый программист. Пишет скрипт для цирка неподалеку, говорит: шарики с водородом - прошлый век, давайте сделаем шарики с гелием. Заменяю - ~~шарик~~ у него гусь из пруда улетел. Две недели разбирается, чтобы понять, почему.

ответить

▲ +68 ▼ 🐼 Vlad528 3 года назад

С водородом подъемная сила выше.

ответить

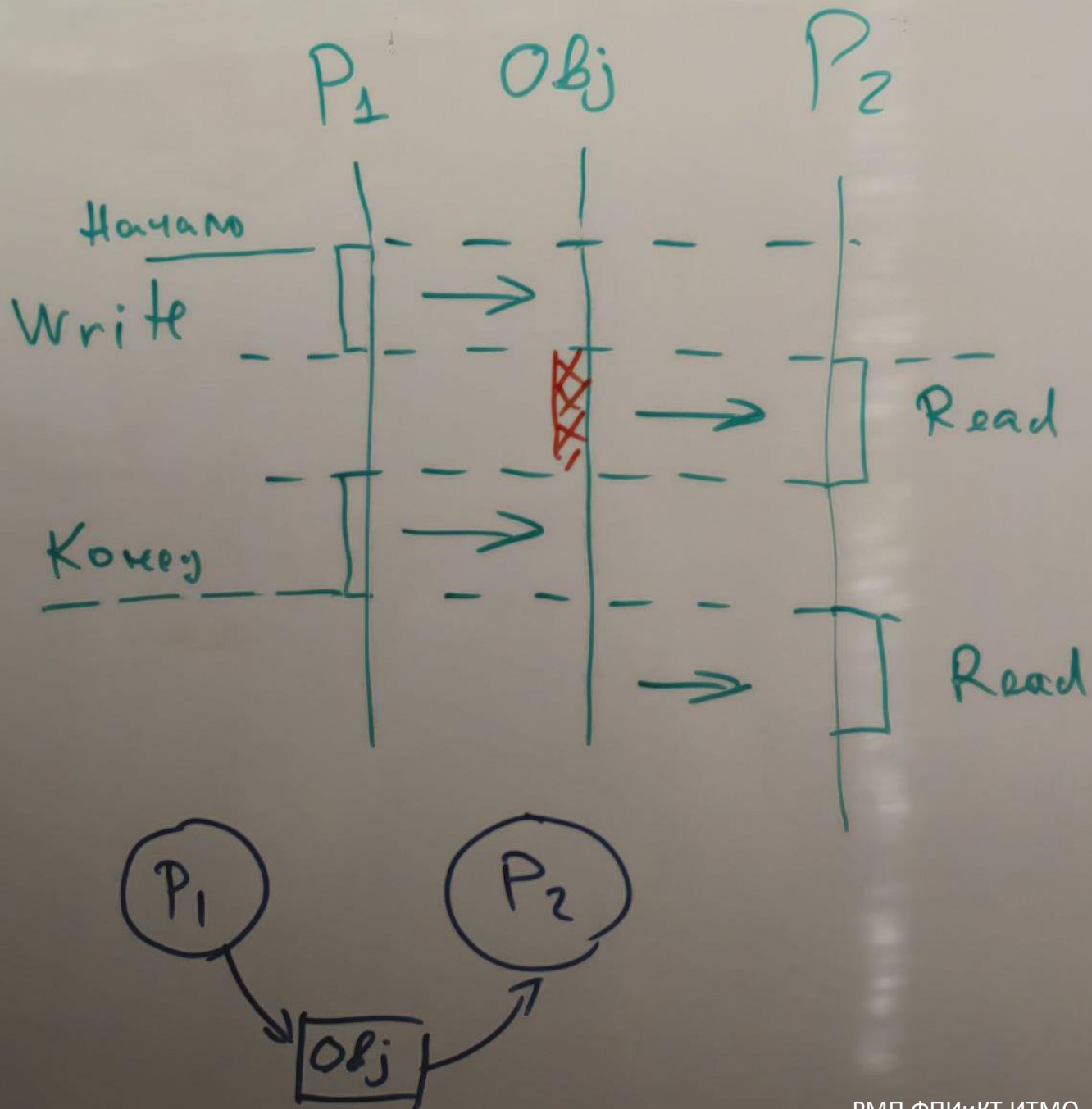
▲ +143 ▼ 🦉 Zmiillo 3 года назад

Так даже эпичнее.

1. Тестировщик говорит об угрозе пожара в цирке.
2. Меняешь водород на гелий в шарах.
3. В соседнем пруду утонул гусь.

Гонки

- Два потока работают с одной и той же структурой данных
- Существует вероятность того, что один поток изменит данные только частично и будет прерван переключателем задач операционной системы.
- Второй поток получает управление от планировщика и получает частично измененные данные.
- Для примера посмотрите реализацию HashMap в OpenJDK:
 - <https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/HashMap.java>
 - Надеюсь, что у вас после этого исчезнет желание использовать экземпляр этого класса из нескольких потоков.



Гонки

Блокировки

- Один поток блокирует данные
- Второй блокирует другие данные, которые нужны первому процессу
- В результате происходит взаимная блокировка навсегда

Способы защиты

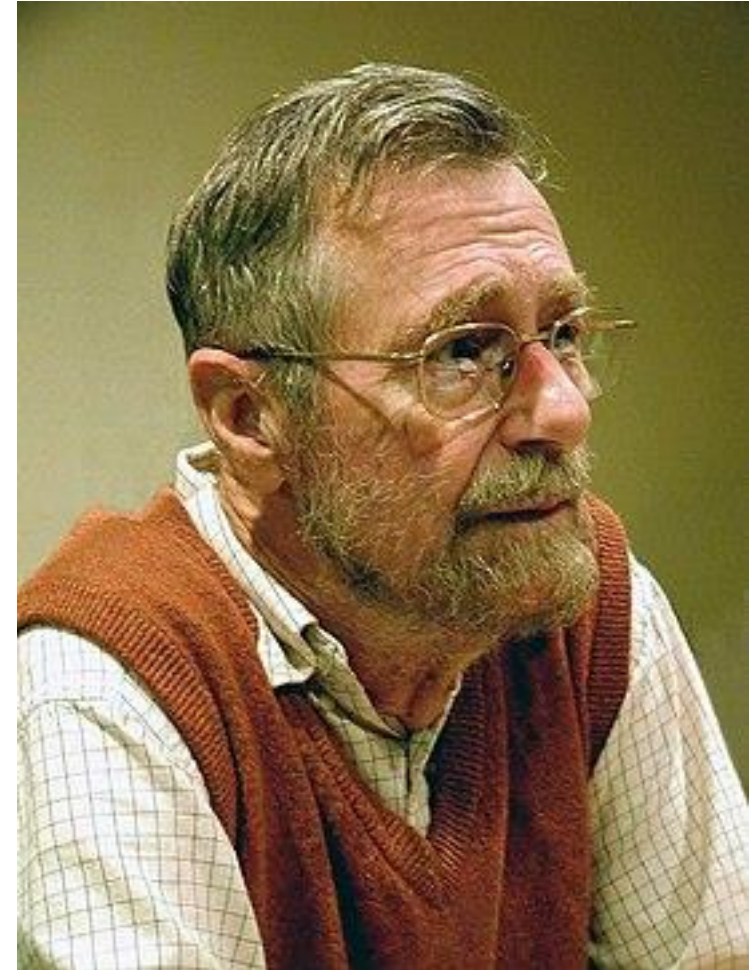
- Критические секции
- Атомарные переменные
- Семафоры Дейкстры

Критические секции

- Критическая секция гарантирует, что в хранилище данных в один момент времени хозяйничает только один поток
- В Java для этого используется `synchronized`
- Как это работает?
 - В примитивных операционных системах просто отключается планировщик на время работы критической секции
 - В Windows критическая секция не дает потокам доступ к защищенному участку кода.
 - **В Linux нет критических секций**, вместо них нужно использовать мьютексы или семафоры
 - В Java `synchronized` защищает часть кода от доступа со стороны других процессов (это примерно аналогично тому, что сделано с критическими секциями в Windows).

Семафоры Дейкстры

- Семафор – блокирующая переменная s , изменение состояния которой производится одним действием (т.е. изменение нельзя прервать) с использованием функций $P(s)$ и $V(s)$.
- Существует два действия над семафорами:
 - $P(s)$ - занять семафор
 - $V(s)$ - освободить семафор.
- При попадании на занятый семафор процесс блокируется до тех пор, пока другой процесс не освободит семафор.

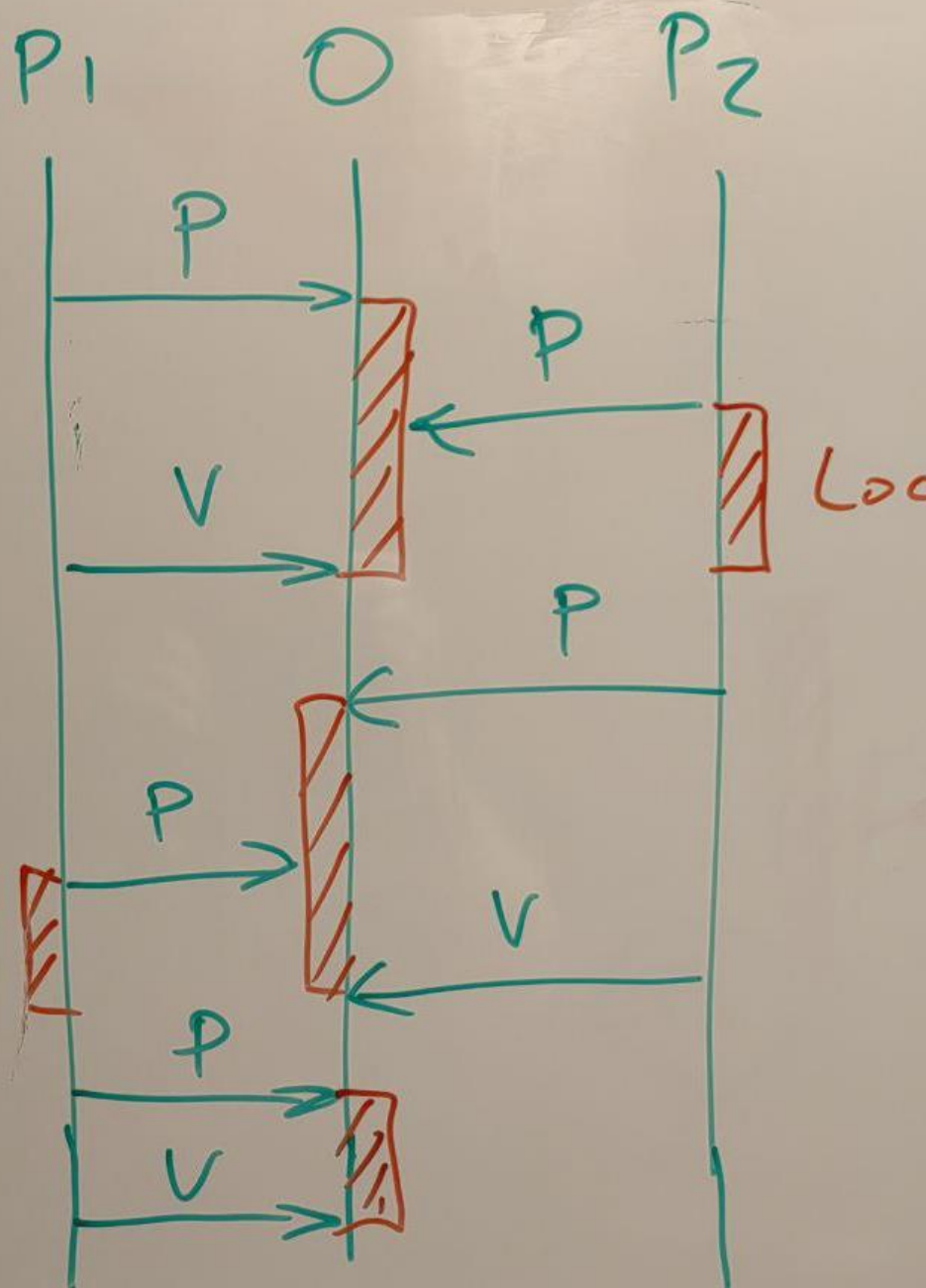
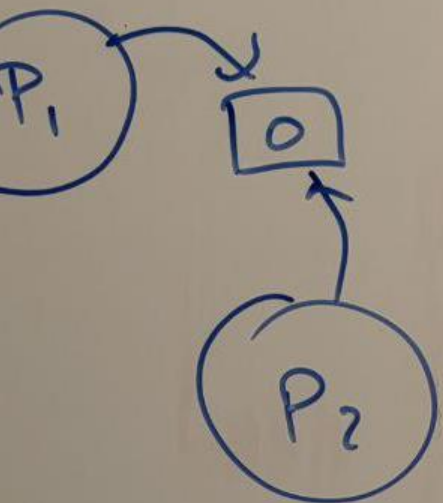


- **Функция P(s)**

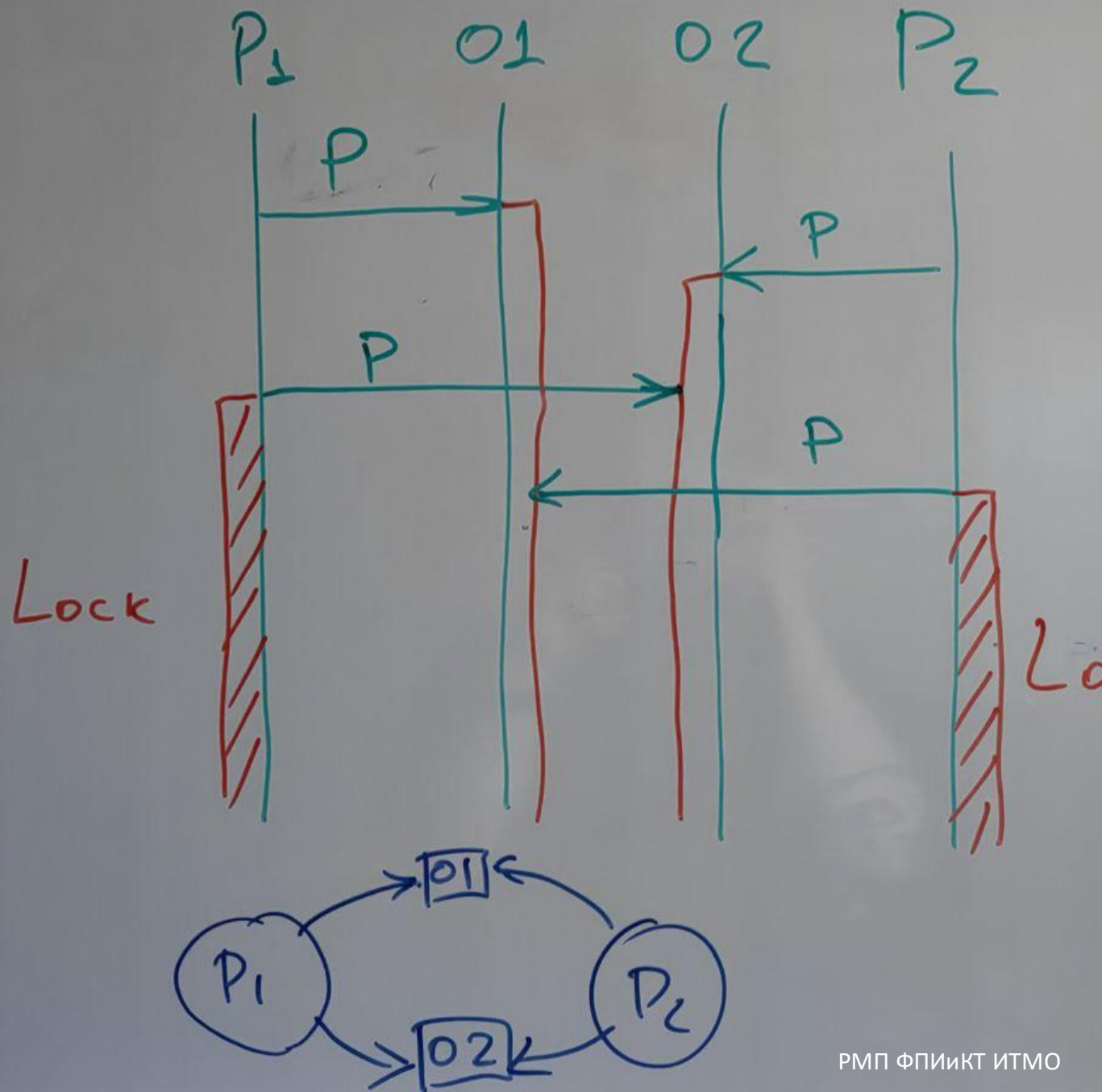
```
if( s == 0)
    Заблокировать_текущий_процесс();
else
    s--;
```

- **Функция V(s)**

```
if( s == 0)
    разблокировать_один_из_процессов();
else
    s++;
```



Семафоры:
Нарру Way,
ожидание



Семафоры: суровая реальность

«Семафор – это элегантный инструмент синхронизации для идеального программиста не допускающего ошибок»
[Бринч Хансен]

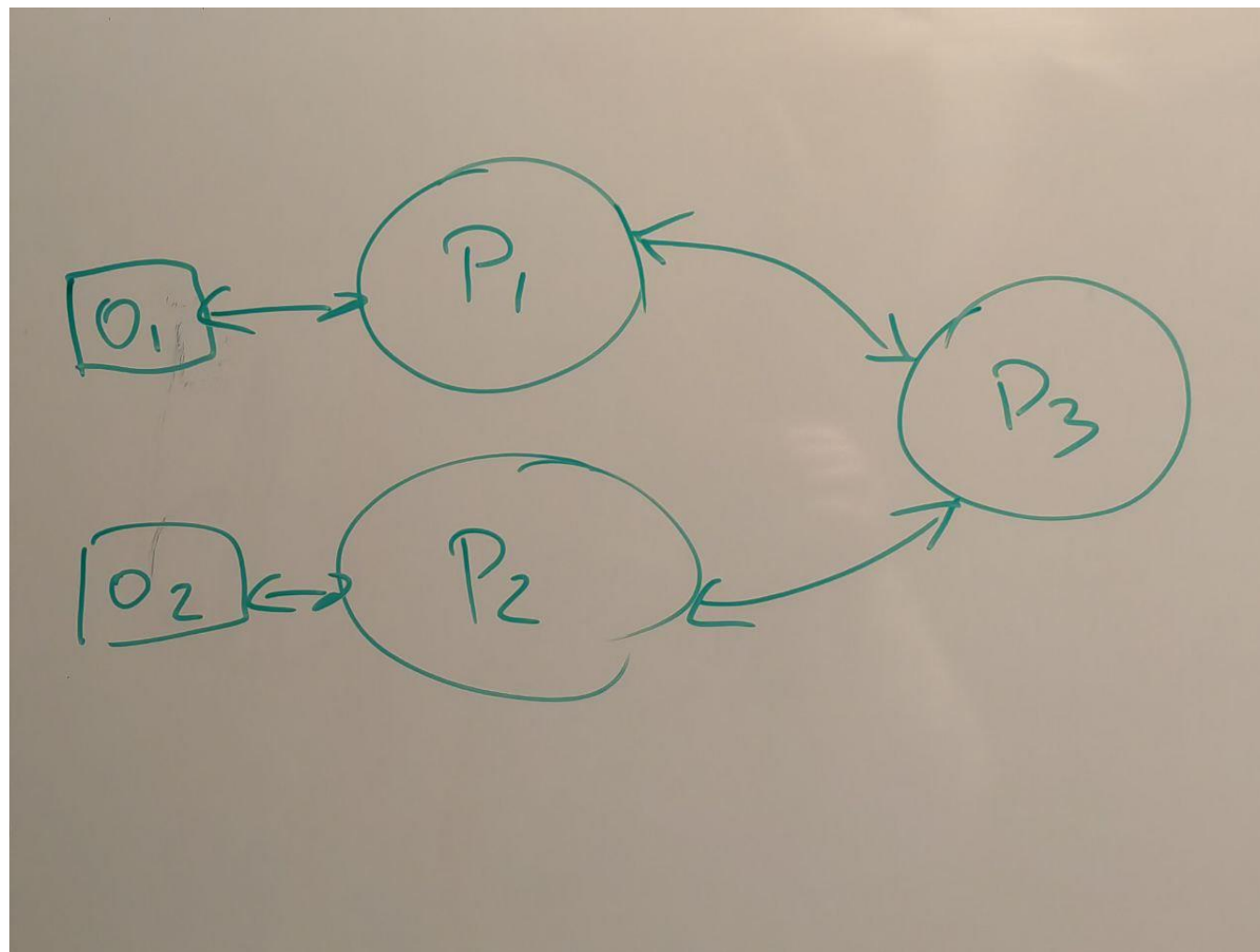
Атомарные переменные

- Атомарные переменные устроены таким образом, чтобы поток мог изменить их значение полностью за один раз.
- См. Java/Kotlin AtomicInteger
 - `var counter : AtomicInteger = AtomicInteger(0)`
 - `counter.incrementAndGet()`

Как получить детерминизм в своей программе?

- Необходимо использовать модели вычислений, которые поддерживают параллелизм.
- Один из вариантов – сети процессов Кана
 - В модели есть только процессы и бесконечные очереди
 - Никаких глобальных переменных или общей памяти нет
 - Процессы независимы друг от друга и не имеют семафоров

Сети процессов Кана



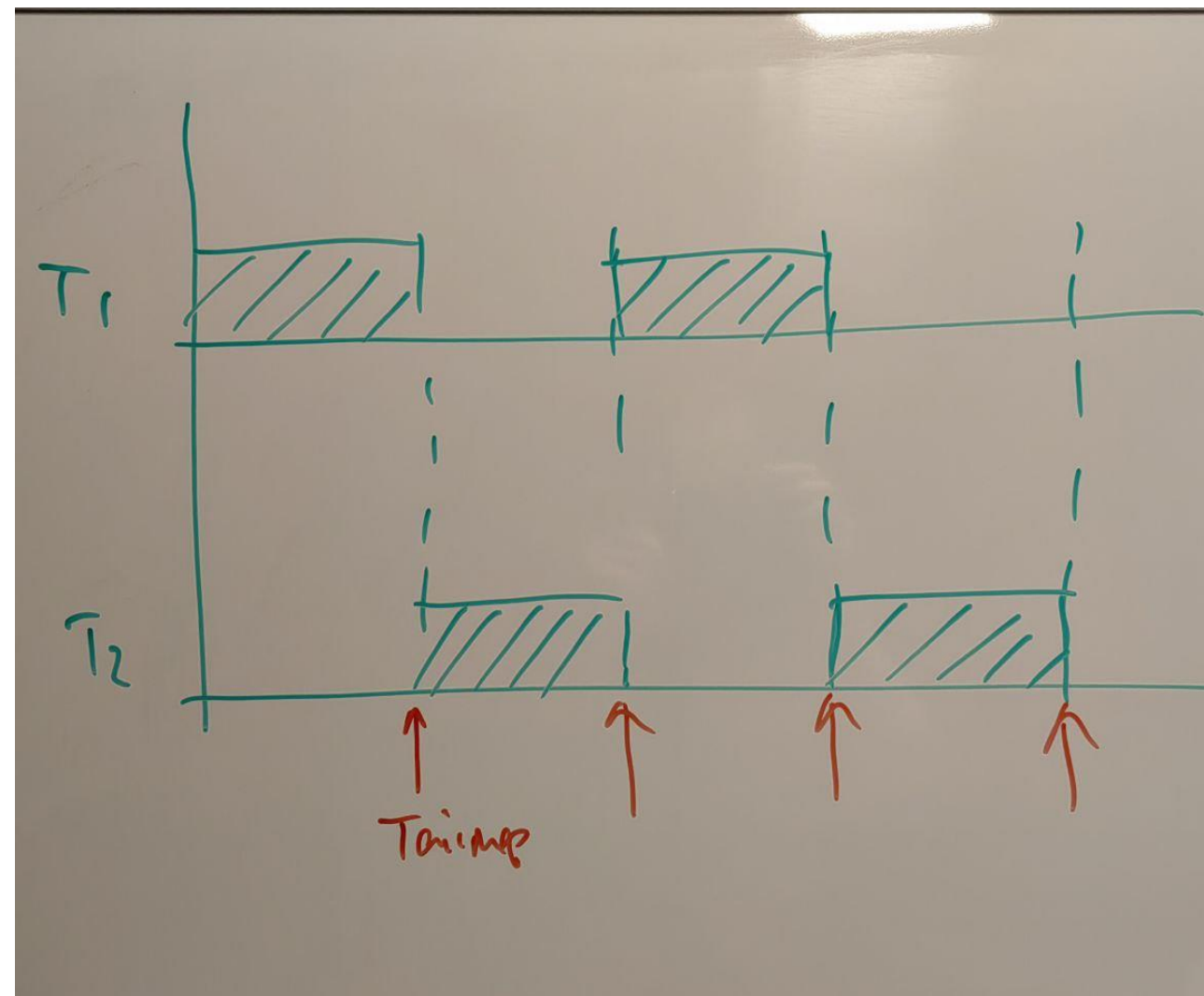
Как реализовать сети процессов в Kotlin

- Потоки и каналы
- Корутины и очереди

Потоки vs корутины Kotlin

- Потоки базируются на потоках Linux, то есть используют вытесняющую многозадачность, в которой переключатель задач прерывает исполнение текущей задачи и передает управление другой, готовой к исполнению задаче.
- Корутины основаны на идее согласующей (кооперативной) многозадачности, когда передача управления происходит не по воле планировщика, а по инициативе самой задачи. В Kotlin, команды для передачи управления другим корутинам спрятаны под капот компилятора и расставляются автоматически (получается, что переключатель задач как бы размазан тонким слоем по прикладной программе).
- Корутины гораздо эффективнее потоков и требуют гораздо меньших ресурсов.

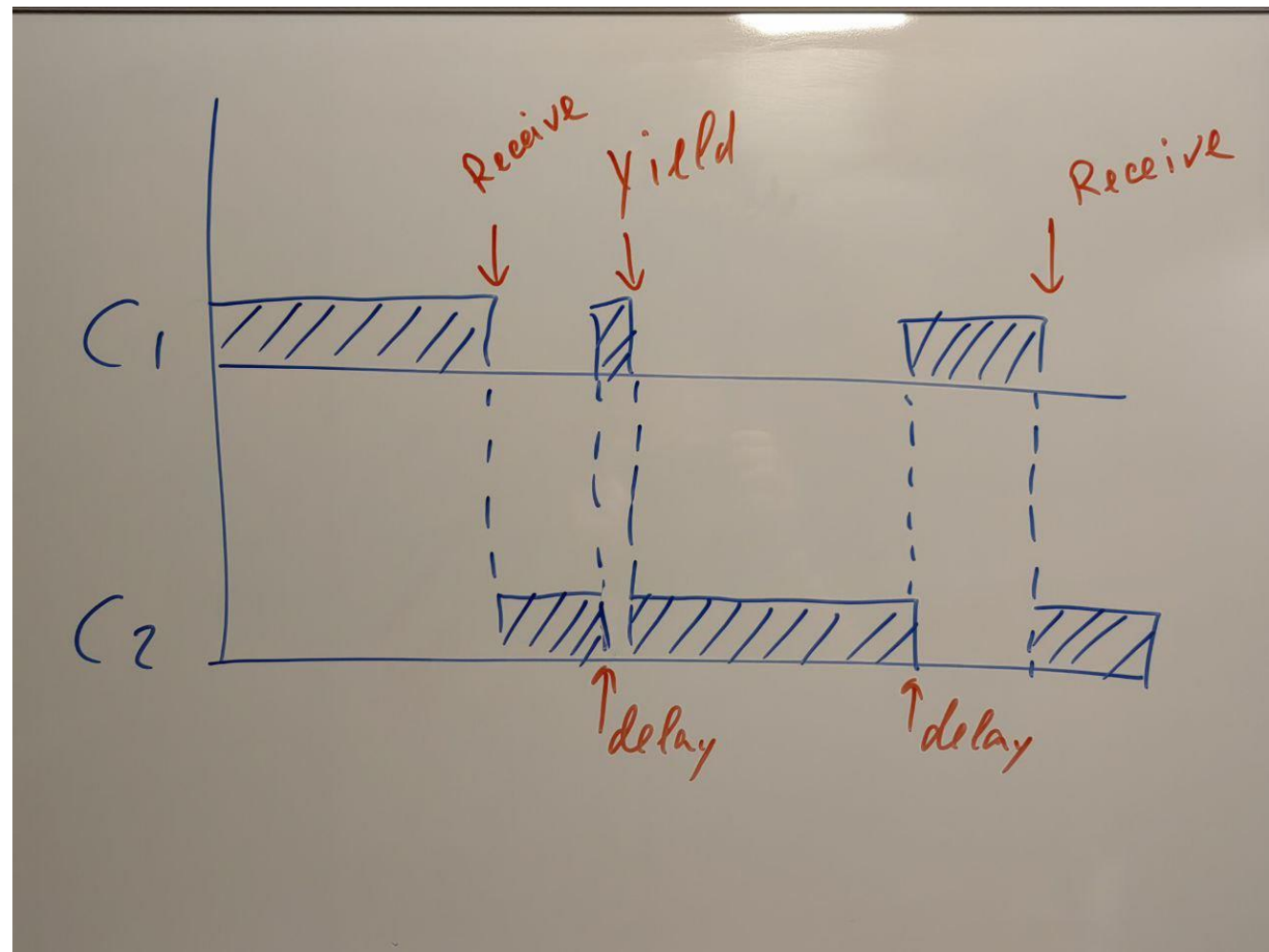
Реализация ПОТОКОВ



Один из вариантов создания потока в Kotlin

```
class ThreadClass3 {  
    fun start() {  
        Thread {  
            repeat( times: 5) { it: Int  
                println("${Thread.currentThread()} run #${it}")  
                Thread.sleep( millis: 500)  
            }  
        }.start()  
    }  
}
```

Реализация коруитн



Корутины Kotlin

- <https://kotlinlang.ru/docs/async-programming.html>
- <https://kotlinlang.ru/docs/coroutines-guide.html>
- <https://github.com/Kotlin/kotlinx.coroutines>
- [Роман Елизаров — Корутины в Kotlin](#)

```
import kotlinx.coroutines.*

fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous one is delayed
}
```


<https://kotlinlang.org/docs/coroutines-basics.html#your-first-coroutine>

Зачем нужны корутины в Kotlin?

- Корутины - ответ на появление высоконагруженных систем и микросервисов с тысячами параллельных процессов. Проблема всплыла примерно в 2010 году.
- Упрощение асинхронного кода
 - Нет ада вложенных обратных вызовов (callback hell)
 - Проще отладка
- Увеличение эффективности кода по сравнению с классическими потоками

WHAT THE HECK IS CALLBACK HELL?

```
2
3
4 a(function (resultsFromA) {
5     b(resultsFromA, function (resultsFromB) {
6         c(resultsFromB, function (resultsFromC) {
7             d(resultsFromC, function (resultsFromD) {
8                 e(resultsFromD, function (resultsFromE) {
9                     f(resultsFromE, function (resultsFromF) {
10                         console.log(resultsFromF);
11                     })
12                 })
13             })
14         })
15     })
16 });
17
```

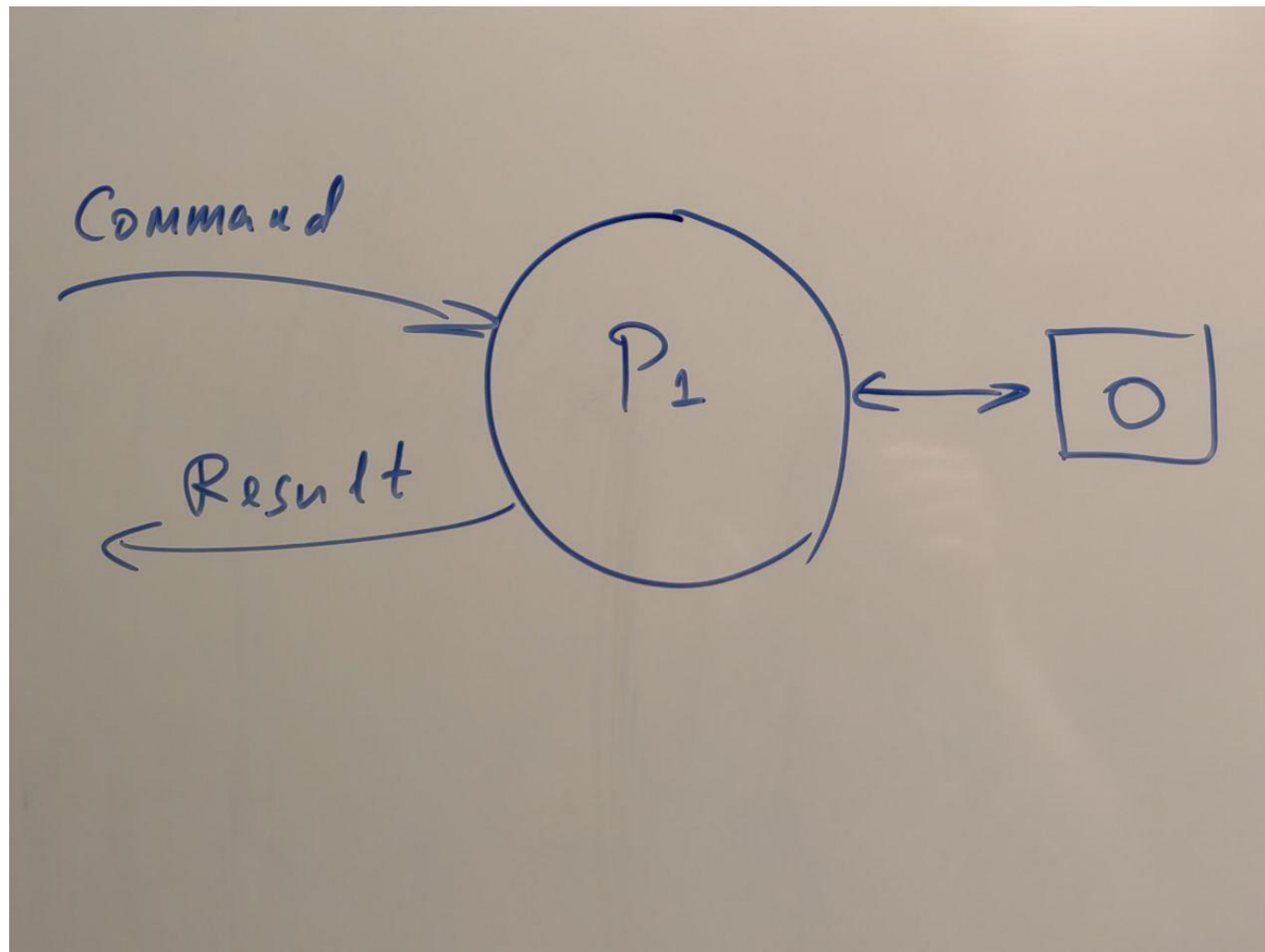


Пример корутины

```
fun foo() {  
    val context = newSingleThreadContext( name: "test")  
  
    CoroutineScope(context).launch { this: CoroutineScope  
        repeat( times: 10 ) { it: Int  
            println("${Thread.currentThread()} run #$it")  
            delay( timeMillis: 50)  
        }  
    }  
}
```


Актор

```
val c = actor {  
    // initialize actor's state  
    for (msg in channel) {  
        // process message here  
    }  
}  
// send messages to the actor  
c.send(...)  
...  
// stop the actor when it is no longer needed  
c.close()
```



Актор

- <https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/actor.html>
- Актор запускает новую сопрограмму, которая получает сообщения из канала почтового ящика и возвращает ссылку на свой канал почтового ящика в качестве канала отправки. Полученный объект может быть использован для отправки сообщений в эту сопрограмму.
- Акторы упрощают создание менеджеров ресурсов, защищающих какие-либо сложные структуры данных от гонок при доступе из нескольких конкурирующих процессов.

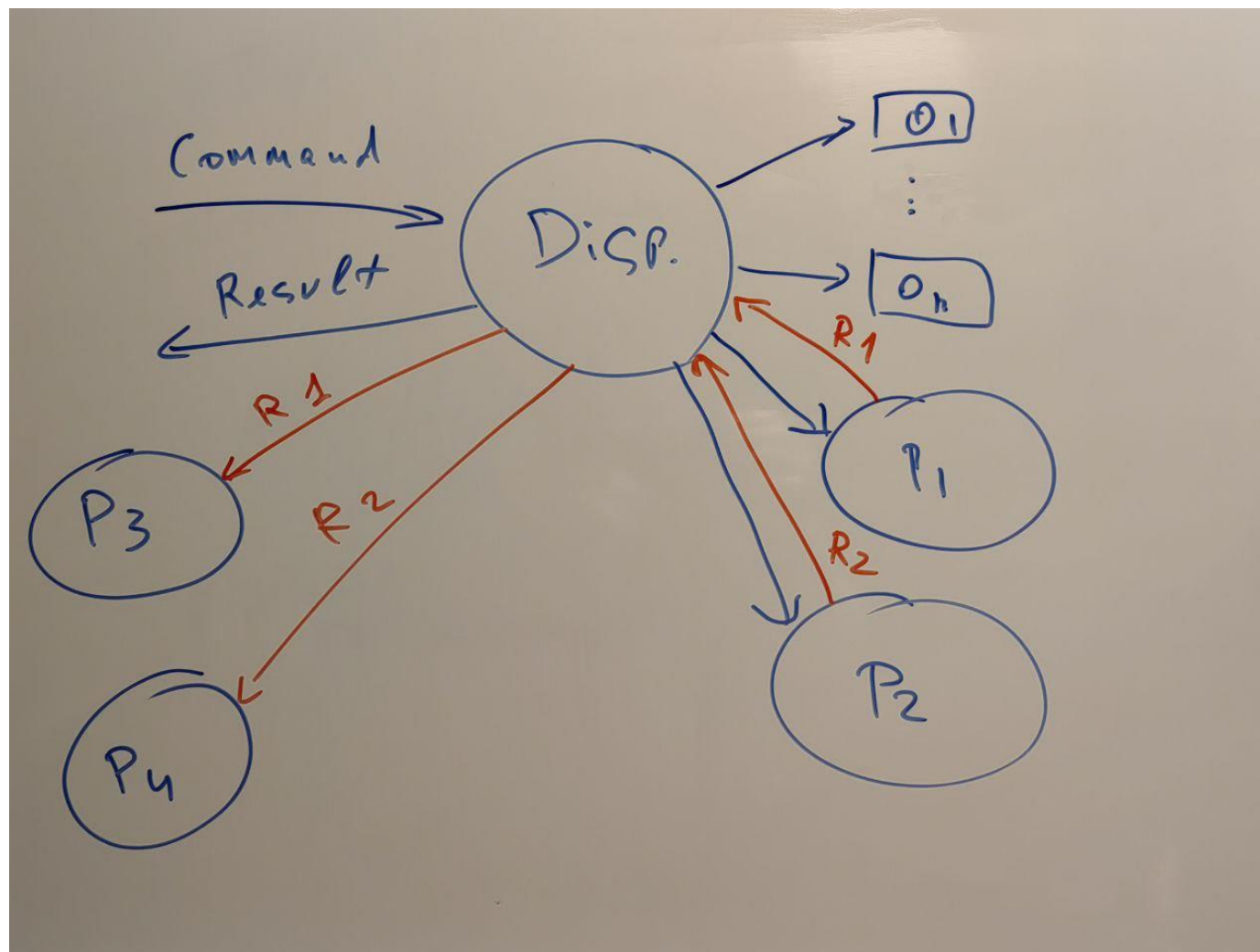
Пример актора #1

```
object ActorExample {  
    private var lnk : SendChannel<LinkMsg>  
    private val scope = CoroutineScope(newSingleThreadContext(name: "TestActor"))  
  
    sealed class LinkMsg  
    private class InitCounter(val newValue: Int = 0) : LinkMsg()  
    private class Write(val value: String) : LinkMsg()  
    private class Read(val response: CompletableDeferred<String>) : LinkMsg()  
  
    init {  
        lnk = scope.linkActor()  
    }  
  
    suspend fun initCounter(newValue : Int = 0) : Unit = lnk.send(InitCounter(newValue))  
    suspend fun write(value: String) : Unit = lnk.send(Write(value))  
    suspend fun read(): String {  
        val response = CompletableDeferred<String>()  
        lnk.send(Read(response))  
        return response.await()  
    }  
}
```

Пример актора #2

```
fun CoroutineScope.linkActor() : SendChannel<LinkMsg> = actor<LinkMsg> {  
    var str = ""  
    var counter = 0  
  
    for (msg in channel) {  
        when (msg) {  
            is InitCounter -> counter = msg.newValue  
            is Write -> str = msg.value  
            is Read -> {  
                val result = "${str}${counter}"  
                msg.response.complete(result)  
                counter++  
            }  
        }  
    }  
}
```

Диспетчер



Диспетчер #1

```
class DispatcherExample(val channel : Channel<Int> ) {  
    val context = newSingleThreadContext( name: "test")  
  
    fun start() {  
        CoroutineScope(context).launch { this: CoroutineScope  
            println("Dispatcher start")  
  
            while( true ) {  
                when( channel.receive() ) {  
                    0 -> phase1()  
                    1 -> phase2()  
                    2 -> phase3()  
                    4 -> {  
                        println("4. PROFIT!!!")  
                        println("Dispatcher stop")  
                        return@launch  
                    }  
                    else -> println("4. ???")  
                }  
            }  
        }  
    }  
}
```

Диспетчер

#2

```
fun phase1() : Job = CoroutineScope(context).launch { this: CoroutineScope
    println("1. Придумать требования")
    delay( timeMillis: 1000)
    println("Требования придуманы")
}

fun phase2() : Job = CoroutineScope(context).launch { this: CoroutineScope
    println("2. Составить ТЗ")
    delay( timeMillis: 1000)
    println("ТЗ составлено")
}

fun phase3() : Job = CoroutineScope(context).launch { this: CoroutineScope
    println("3. Разработать архитектуру")
    delay( timeMillis: 1000)
    println("Архитектура разработана")
}
```