Операционни системи за реално време

Автор: доц. д-р инж. Любомир Богданов

Европейски съюз

ПРОЕКТ BG051PO001--4.3.04-0042
„Организационна и технологична инфраструктура за учене през целия живот и развитие на компетенции”
Проектът се осъществява с финансовата подкрепа на
Оперативна програма „Развитие на човешките ресурси”,
съфинансирана от Европейския социален фонд на Европейския съюз
Инвестира във вашето бъдеще!

ЕСФ България
Европейски социален фонд

Съдържание

Схеми за диспечериране на задачите

Операционна система за реално време (**R**eal-**T**ime **O**perating **S**ystem, RTOS) – фърмуер, с помощта на който може да се изпълняват повече от една “main” функции на един микропроцесор и който осигурява стандартни библиотеки, характерни за по-мощни системи (като персонални компютри).

Ако в микроконтролера е вграден един микропроцесор, изпълнението на “main” програмите е **псевдопаралелно**.

Процеси (process) - програми, които се изпълняват в паралел от RTOS.

Нишки (threads) *(или още - задачи, tasks)* – програми, които се стартират от един процес и работят (псевдо)паралелно във виртуалното адресно поле /ако има MMU/ на същия този процес.

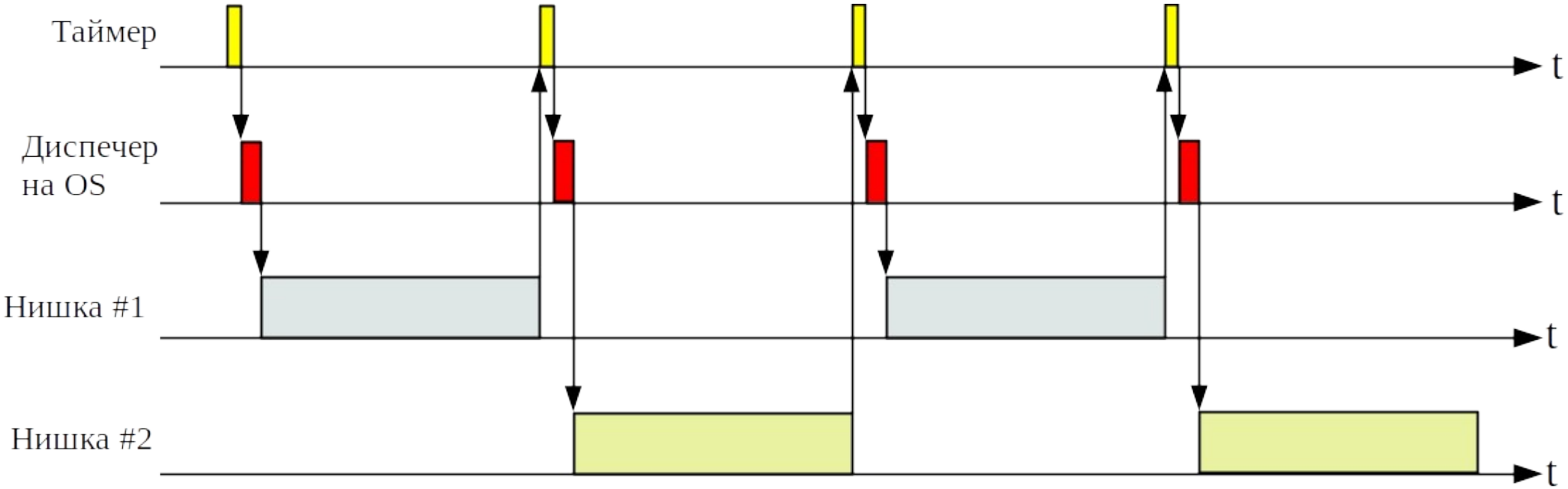
Схеми за диспечериране на задачите

В микроконтролери без MMU понятието процес и нишка съвпадат.

Диспечер на операционната система (scheduler) – част от кода на RTOS, която е отговорна за даване на процесорно време на всеки един процес. Действието на превключване от един процес към друг се нарича **контекстно превключване** (context switch). Кодът на диспечера се изпълнява от същия микропроцесор, който изпълнява процесите.
Да не се бърка с диспечер на инструкцията, който е хардуерен модул от μPU ядро.

Квантовете от време се задават от един хардуерен таймер. Прекъсванията му прехвърлят процесорното време към диспечера.

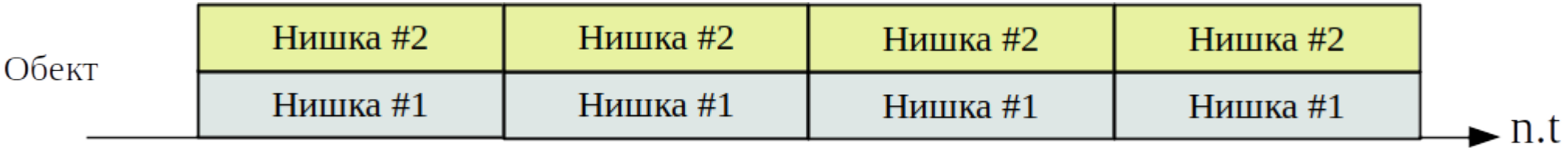
Схеми за диспечериране на задачите



Схеми за диспечериране на задачите

Ето какво “вижда” µPU:

Ето какво “вижда” управляваният обект (вижте времевата база n.t >> t):



или с други думи, ако диспечерът превключва управлението на нишки много бързо, така че управляваният обект да не забележи, за обекта ще е еквивалентно все едно два микропроцесора изпълняват две програми – процес 1 и процес 2.

Схеми за диспечериране на задачите

Въпросът е – колко бързо трябва да става това превключване (time slice)?

Схеми за диспечериране на задачите

Отговорът е – зависи.

Зависи от управлявания обект. Може да е:
*x100 ns
*(x1 ÷ x100) μs
*(x1 ÷ x100) ms
*(x1 ÷ x100) s

За голяма част от некритичните приложения 10 ÷ 100 ms е достатъчно време.

Реално време – означава, че нишките трябва да завършат дадена операция в рамките на предварително известно време.

Схеми за диспечериране на задачите

Операционна система за критично реално време (hard real-time operating system) – ако дадената операция не завърши в даден отрязък от време, ще има катастрофални последници (влак ще спре да се движи, кола ще спре да завива, кран ще зависне и т.н.).

Операционна система за некритично реално време (soft real-time operating system) - ако дадената операция не завърши в даден отрязък от време, няма да има катастрофални последници (ще излезе син екран на кафе машината, ще се рестартира графичния интерфейс на пералнята, ще се забави изпращането на Интернет пакет и т.н.).

Схеми за диспечериране на задачите

Нишките минават през три етапа по време на изпълнението си:

*__блокирана__ (wait) – нишката не се изпълнява от диспечера, защото чака някакво условие да се изпълни;

*__изпълнява се__ (run) – нишката не е блокирана и се изпълнява от диспечера в нейният си времеви квант;

*__готова за изпълнение__ (ready) – нишката не е блокирана, но чака разрешение от диспечера, за да продължи да се изпълнява.

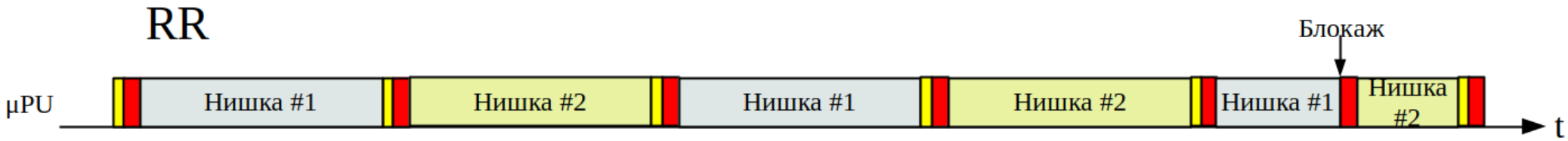
Схеми за диспечериране на задачите

Схема за диспечериране на нишките (scheduling policy) – политика, която указва кога диспечера да даде микропроцесорно време на даден процес.

Най-често използваните схеми са:
*round-robin (RR)
*preemptive (PRE)
*round-robin/preemptive (RR/PRE)
*cooperative (COOP)

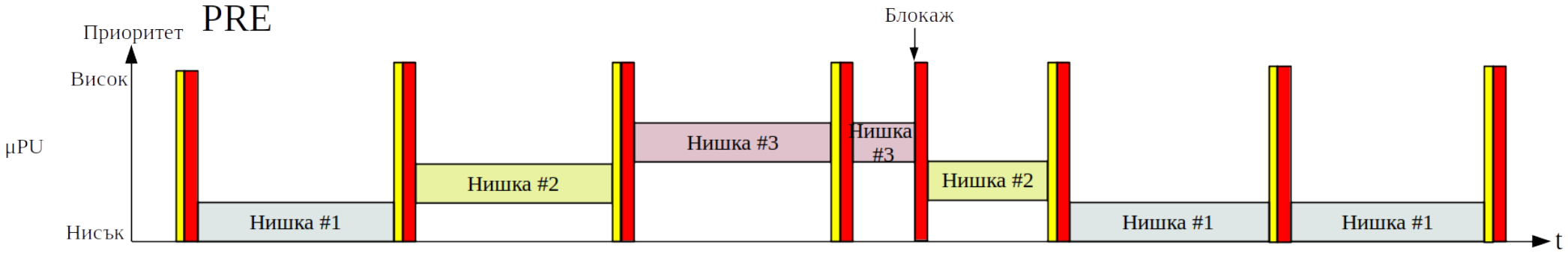
Схеми за диспечериране на задачите

Round-robin диспечериране (RR) – всички нишки са с един приоритет и получават равни квантове време за изпълнение. Ако някоя от нишките блокира, изпълнението се предава на следващата поред.



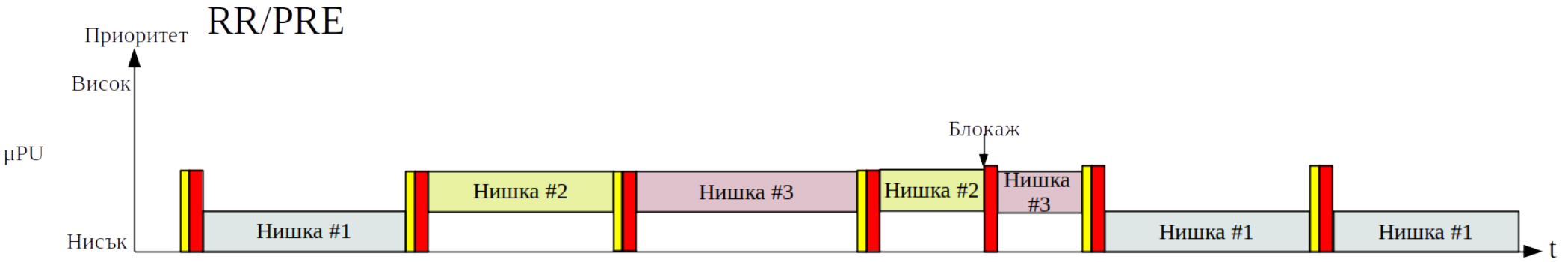
Схеми за диспечериране на задачите

Preemptive диспечериране (PRE) – на всяка нишка се дава уникален приоритет. Не може да има две нишки с един и същи приоритет. Нишки с по-високи приоритети могат да прекъсват (preempt) нишки с по-ниски приоритети. Може да доведе до “приоритетен глад” (priority starvation), където нишките с ниски приоритети почти не се изпълняват заради нишки с по-високи приоритети.



Схеми за диспечериране на задачите

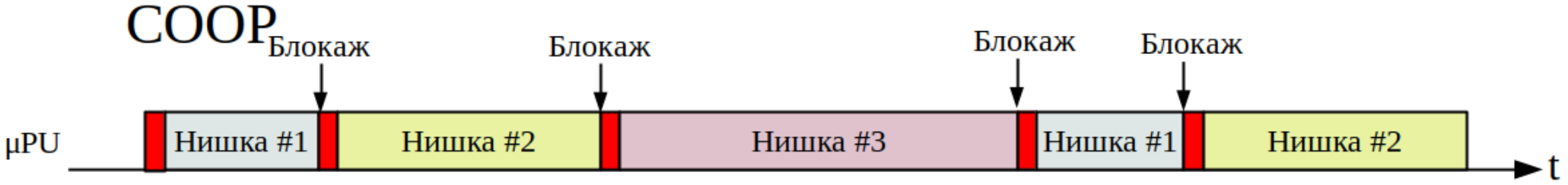
Round-robin/Preemptive диспечериране (RR/PRE) – на всяка нишка се дава приоритет. Може да има две нишки с един и същи приоритет. Нишки с различни приоритети се изпълняват по Preemptive схемата. Нишки с еднакви приоритети се изпълняват по RR схемата.



Пример: #2 и #3 са с един и същ приоритет, а #2 и #3 са с по-голям приоритет от #1

Схеми за диспечериране на задачите

Cooperative диспечериране (COOP) – всички нишки са с един и същ приоритет. Не се използва системен таймер, т.е. превключването от една нишка в друга става без прекъсване от таймер. Разчита се, че всяка една нишка ще пуска следващата като се самоблокира “доброволно”.



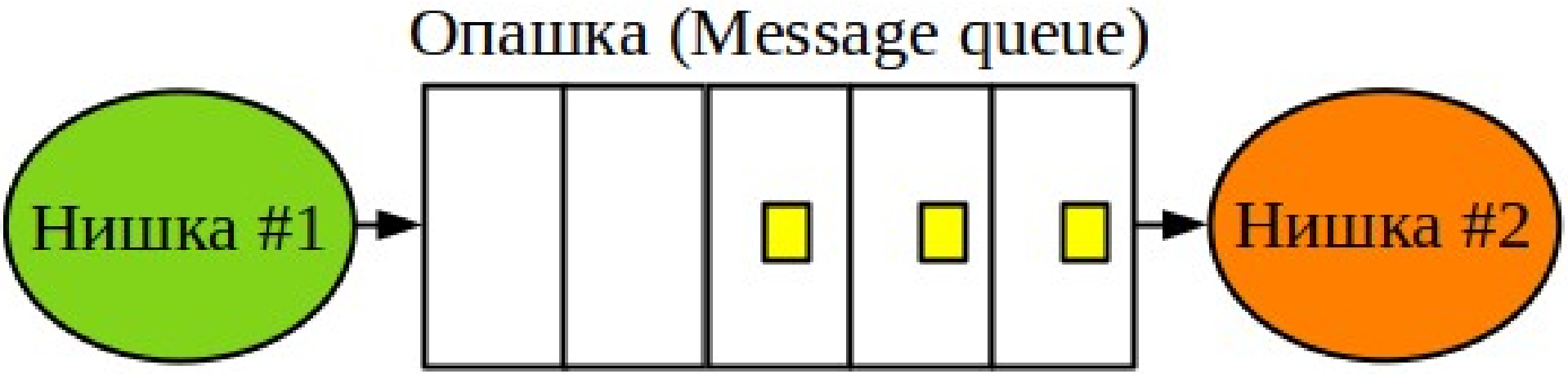
Комуникационни примитиви

Комуникационни примитиви (interthread communication) – променливи, които се използват за предаване на данни от една нишка на друга под управлението на ядрото на RTOS.

Най-често използваните ком. примитиви са:
*опашки
*поща

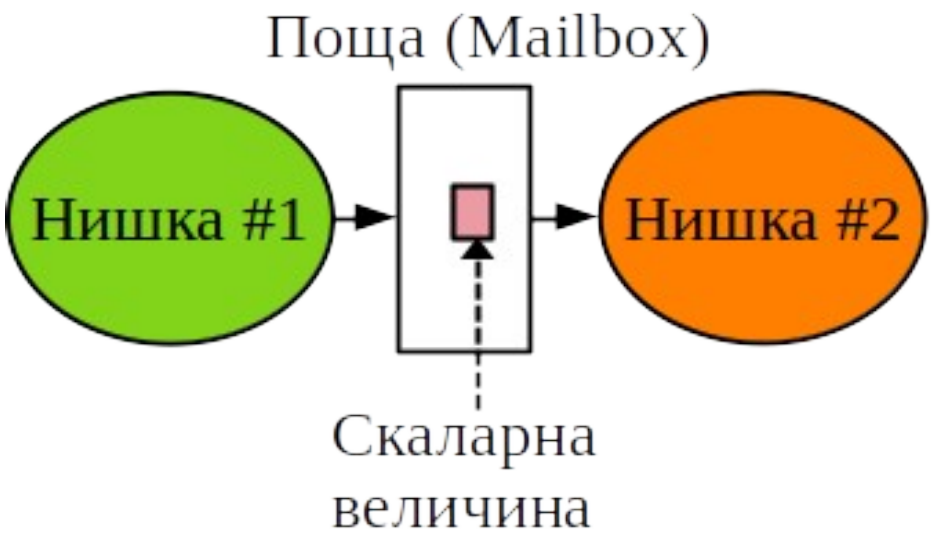
Комуникационни примитиви

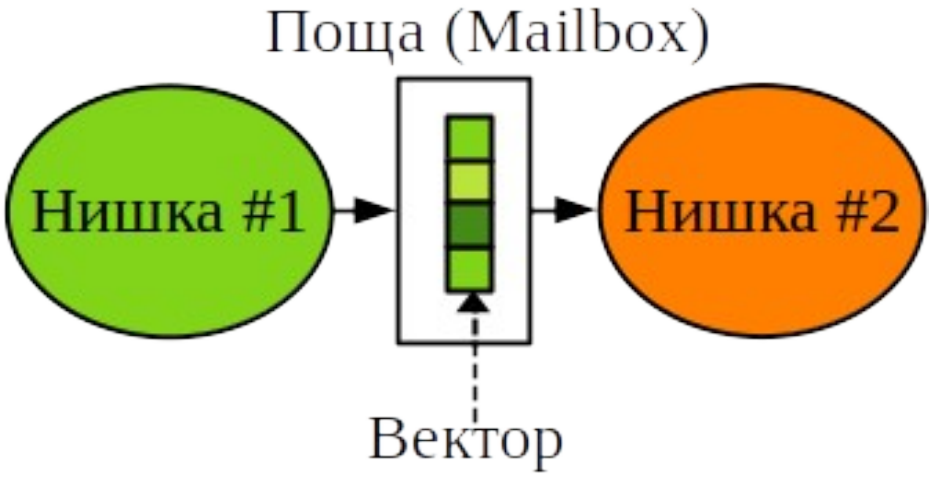
Опашки (message queue) – софтуерни FIFO буфери, които съдържат скаларни величини (int, float, double, char, и т.н.). Те се създават, четат и записват с API функции на ядрото на RTOS [1].



Комуникационни примитиви

Поща (mailbox) – променлива (скаларна величина или вектор), която се използва за комуникация между две нишки. Може да се оприличи на опашка със само един елемент.





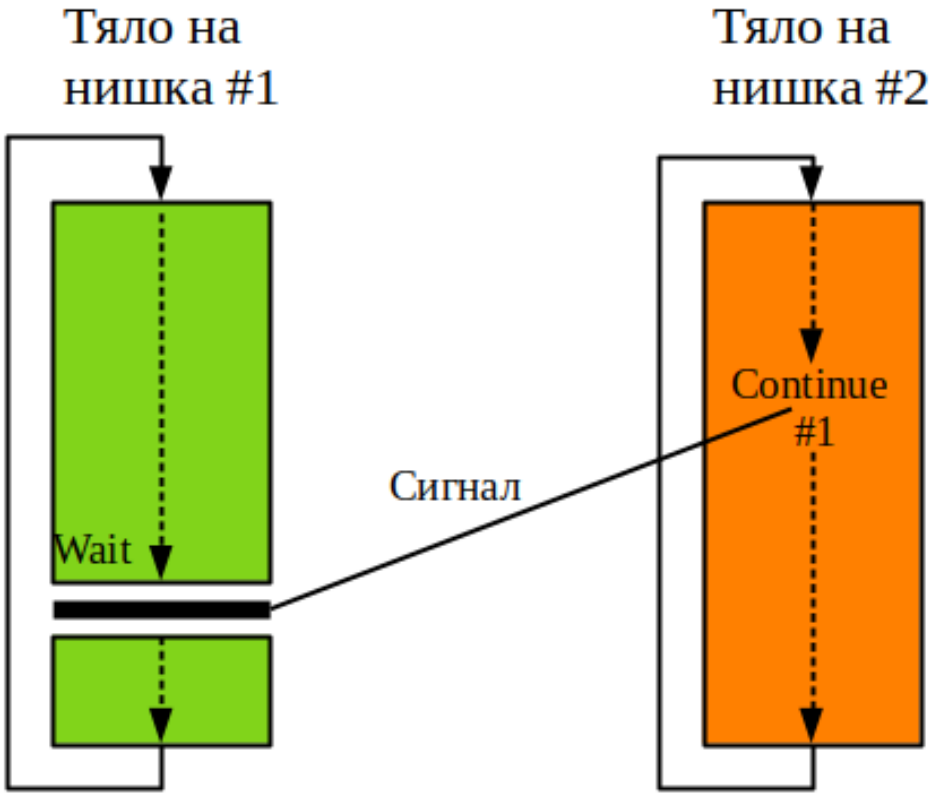
Синхронизационни примитиви

Синхронизационни примитиви (synchronization primitives) променливи, които се използват за синхронизиране изпълнението на една нишка спрямо друга под управлението на ядрото на RTOS [1], [2].

Най-често се използват:
*сигнали
*семафори
*мютекси (двоични семафори)
*рандеву
*бариера

Синхронизационни примитиви

Сигнал (signal) – бит от променлива, на който една нишка може да чака (wait). Когато битът се промени, нишката продължава напред (running). Битът се променя от втора нишка.

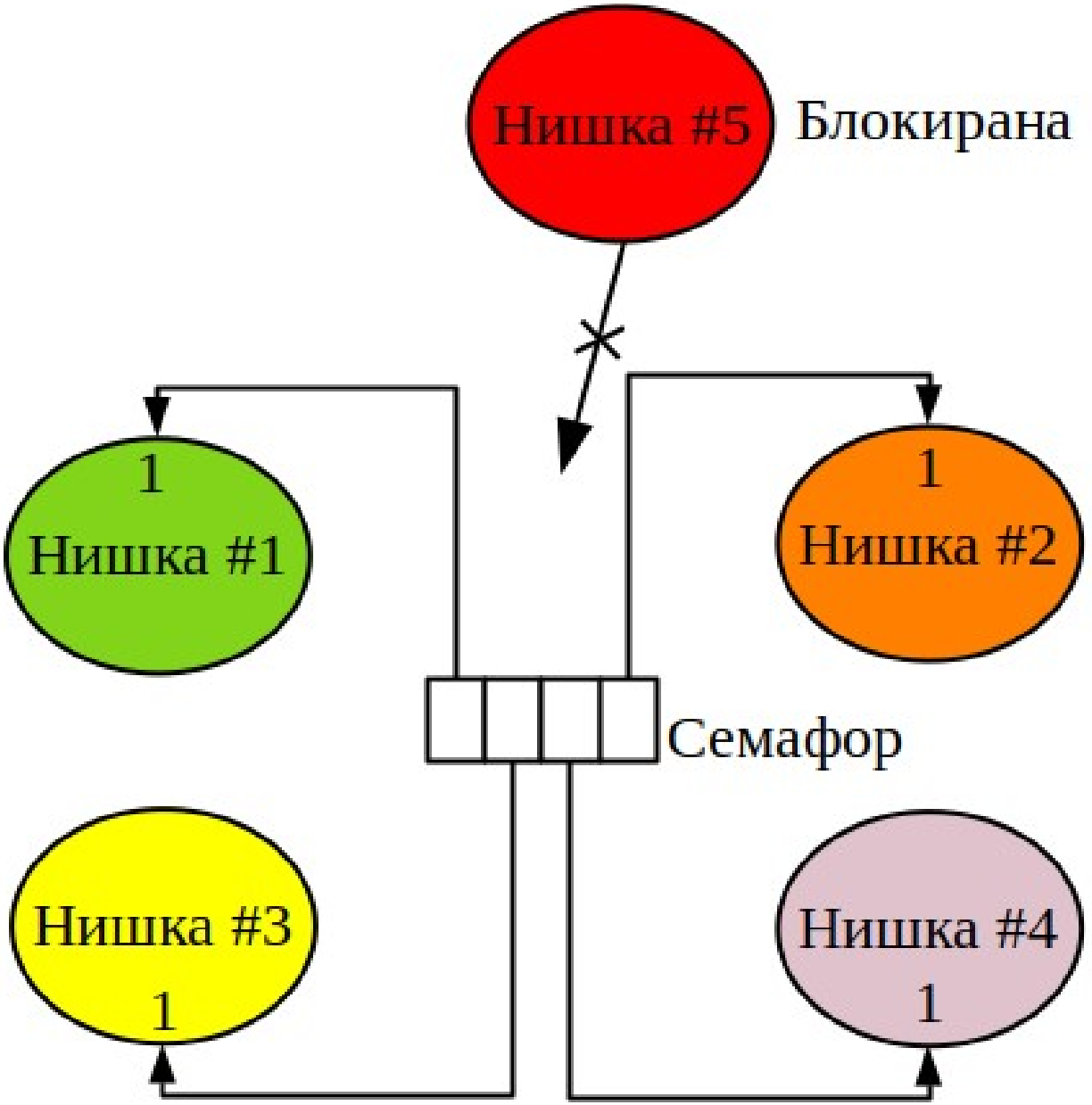


Синхронизационни примитиви

Семафор (semaphore) – променлива, която се зарежда с число от програмиста. Когато една нишка се изпълнява, в тялото на кода ѝ може да се извика функция, която достъпва тази променлива. Ако тя не е нула, нишката ще продължи напред. Ако тя е нула, нишката ще блокира, докато някоя друга нишка не върне число обратно в семафора.

Използва се, за да се контролира достъпа до ограничени ресурси. Например – 4 портова SRAM памет ще може да се достъпи от максимум 4 процеса наведнъж. Ако се появи 5-ти процес, той трябва да бъде блокиран, докато някой от другите не спре да използва паметта.

Синхронизационни примитиви

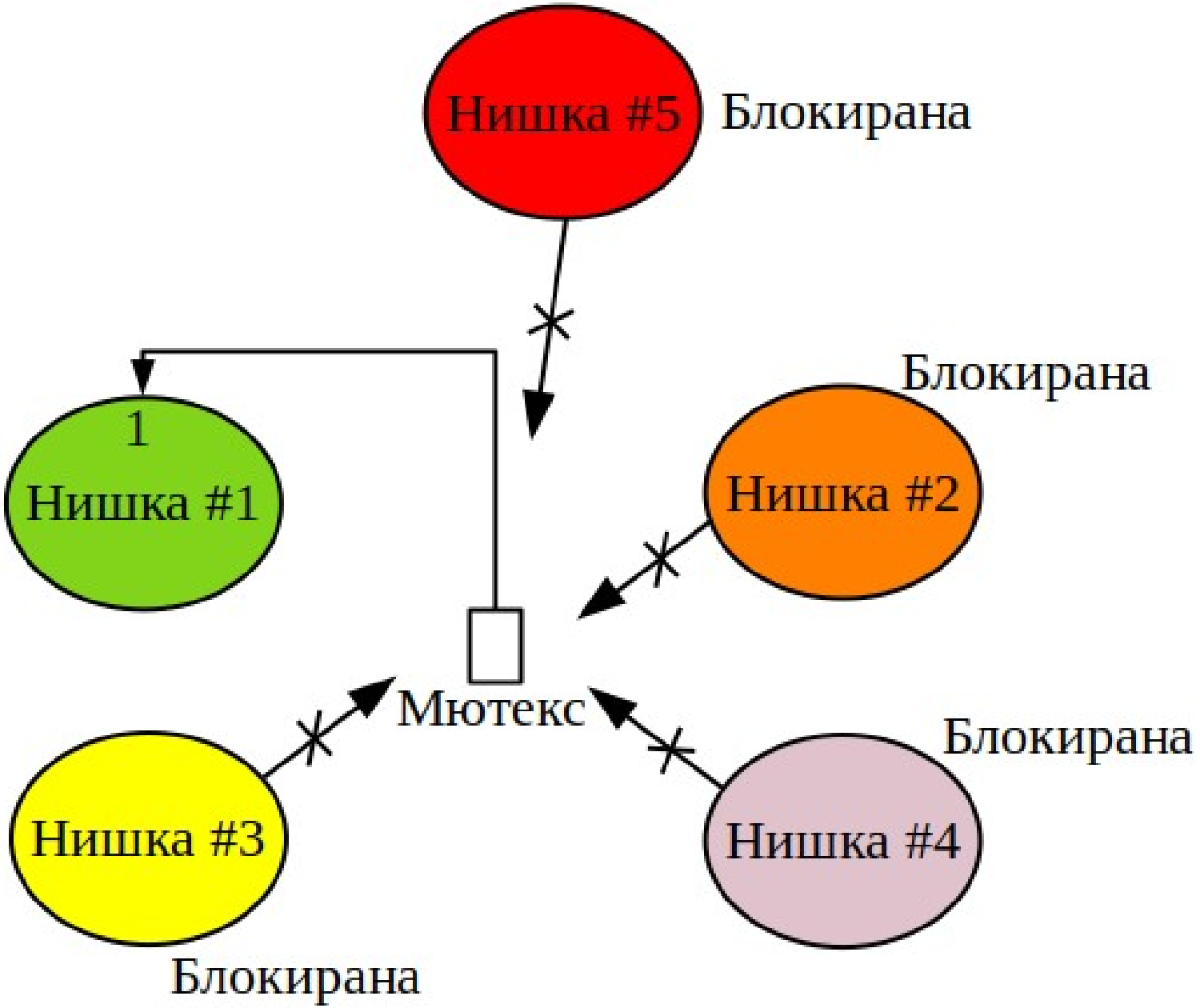


Синхронизационни примитиви

Мютекс (mutex) – променлива, която се зарежда с число, което може да е само 0 или 1, от програмиста. Действието му е аналогично на семафора.

Използва се, за да се контролира достъпа до едноканални/еднопортови ресурси. Например – UART интерфейса може да изпраща съобщение само от една нишка, защото има само един сигнал TxD и изпраща данните серийно.

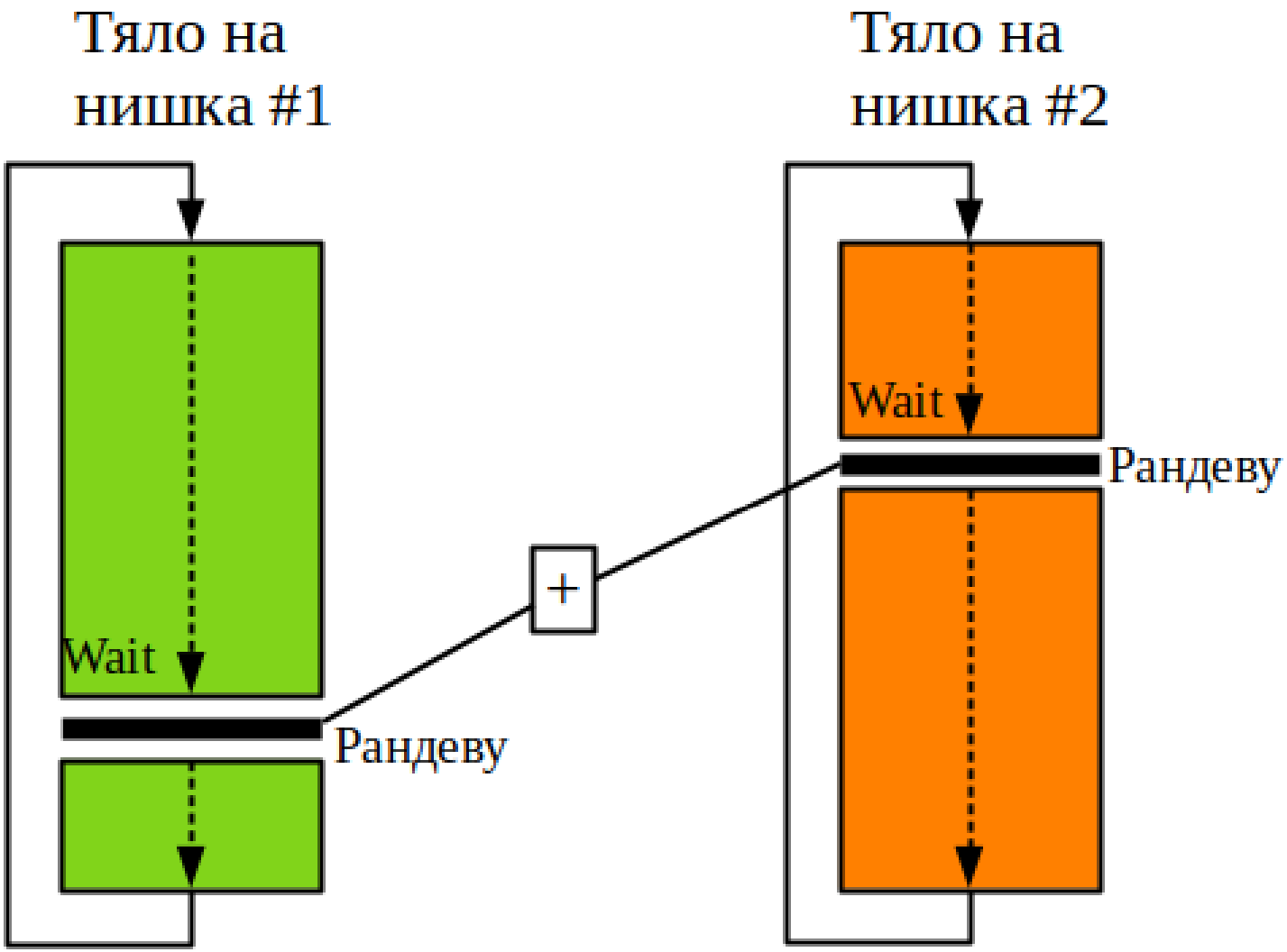
Синхронизационни примитиви



Синхронизационни примитиви

Рандеву (rendezvous) – променлива, която гарантира, че две нишки ще се синхронизират и ще продължат изпълнението на дадени части от кода си едновременно. Може да се направи аналогия с двувходово И – кодът на нишка 1 ще продължи изпълнението си от адрес N, когато нишка 2 стигне до адрес M от кода си.

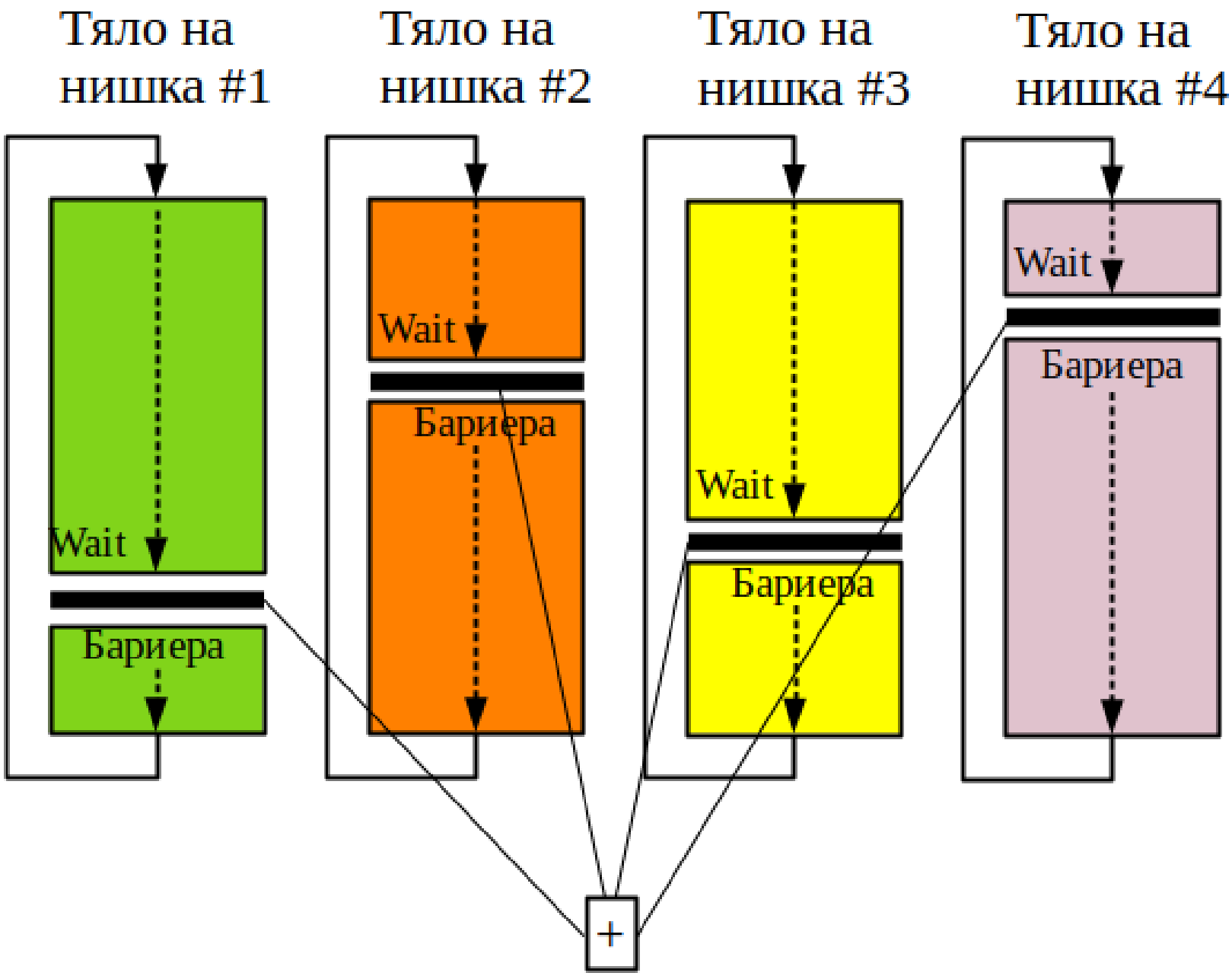
Синхронизационни примитиви



Синхронизационни примитиви

Бариера (barrier turnstile) – променлива, която е по-генерализиран вариант на рандевуто и гарантира, че две или повече нишки ще се синхронизират и ще продължат изпълнението на дадени части от кода си едновременно. Може да се направи аналогия с многовходово И – кодът на нишка 1 ще продължи изпълнението си от адрес N, когато нишка 2 стигне до адрес M от кода си, и нишка 3 до адрес P, и нишка 4 до адрес Q, и т.н.

Синхронизационни примитиви



Линукс за вградени системи

През 1991 г. Линус Торвалдс (Финландия), на 21 г., създава ядро (kernel) за операционна система за IBM компютри, базирани на 80386. По-късно OS се портва и на други микропроцесори.

Ядрото е добавено към операционната система GNU (всичко друго, текстови редактор/компилатор/файлов експлорър/браузъри/командни редове, и т.н. освен ядро). Така се появява името GNU/Linux. Идеята е да се направи алтернатива с отворен код на операционната система Minix (която наподобява Unix).

Въпреки, че много прилича на Minix/Unix, Торвалдс е написал ядрото от 0.

Линукс за вградени системи

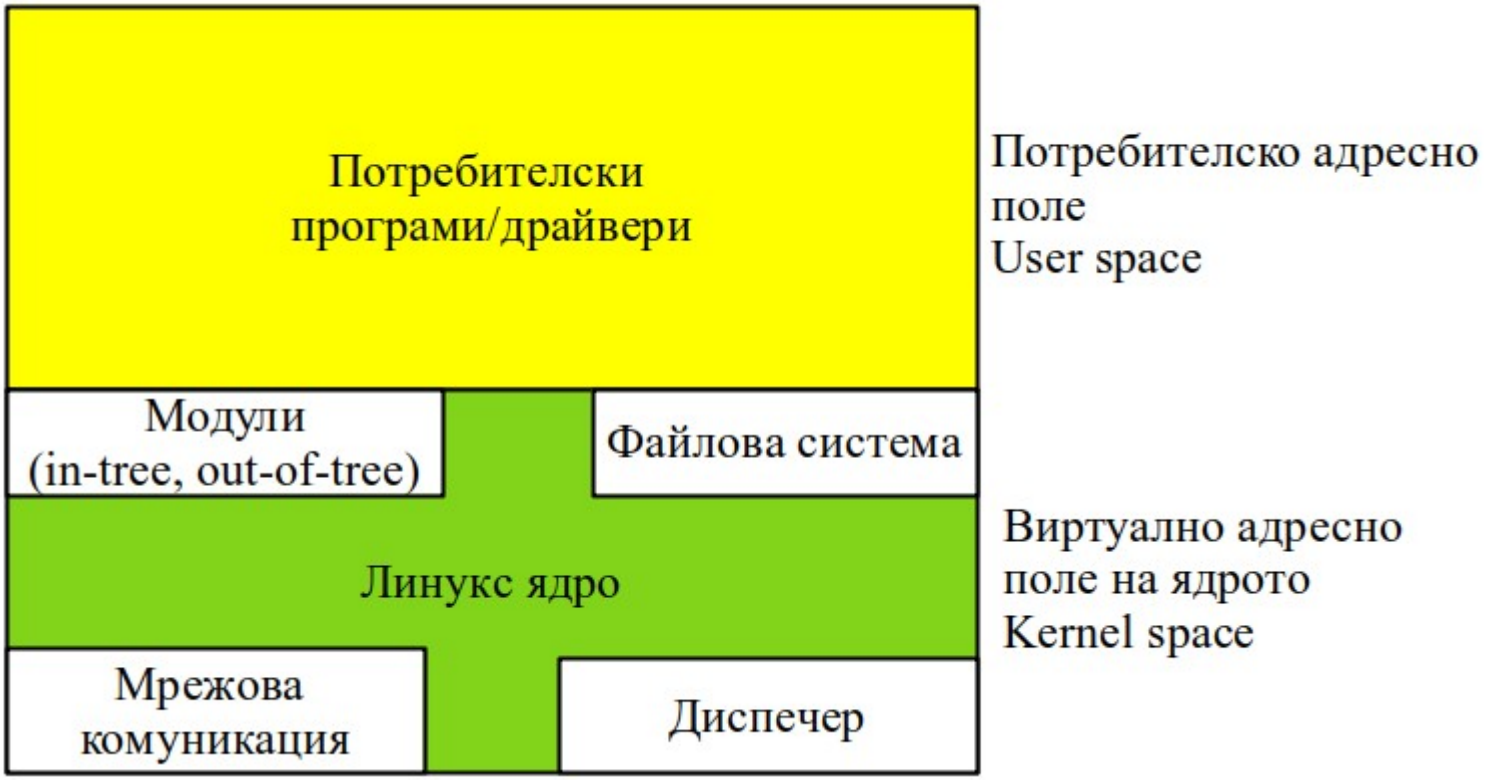
В по-голямата си част Линукс е **POSIX съвместим,** но не е POSIX сертифицирана (POSIX, IEEE 1003, е стандарт за операционни системи, дефиниращ формата на системните интерфейси, т.е. API функции, команден ред, поддържани команди, C библиотеки и др.).

Линукс е **монолитна OS** – това означава, че драйверите (наричат се модули в Линукс), файловите системи и библиотеките за мрежова комуникация са част от виртуалното адресно поле на ядрото. Изброените софтуерни компоненти и диспечер се компилират като един обектов файл с големина 2 ÷ 15 MB.

Линукс позволява **динамично зареждане на модули** – т.е. по време на работа на системата драйверите може да се включват и изключват при необходимост.

Линукс за вградени системи

Линукс използва **preemtive диспечериране** на задачите.



Линукс за вградени системи

Линукс за вградени системи, който се пуска на приложни процесори (Application Processors / SoC) не се различава съществено от *Линукс* за персонален компютър. Често дори има графичен интерфейс.

Линукс за вградени системи

Линукс за вградени системи, който се пуска на микроконтролери с общо предназначение (deeply embedded microcontrollers) има някои забележими разлики:
*няма графичен интерфейс;
*може да не използва MMU;
*може да няма файлова система и възможност за интернет комуникация;
*може да работи от Flash – виж XIP (.text сегмента на ядрото да е във Flash, а .data и .bss – в RAM, докато при PC - .text, .data, .bss са в RAM);
*не използва BIOS, а **дървесни двоични описания** (**device tree binaries**), за да "опознае" системата, на която се изпълнява;

*дистрибуцията с приложни програми е минималистична (напр. BusyBox);
*използва минималистични програми за начално зареждане (bootloaders) като U-boot;
*стартира за 2/3 секунди от 216 MHz-ов микропроцесор с 16 MB RAM.

Линукс за вградени системи

От гледна точка на програмиста за вградени системи, най-важни са модулите (драйверите).

За да се разработват модули за *Линукс* във вградени системи, трябва първо успешно да се създаде (build-не) *Линукс* ядро.

Това става с помощта на cross-toolchain за съответния микропроцесор.

Линукс за вградени системи

От гледна точка на build системата, има два вида модули:

*__в сорс-дървото__ на Линукс (in-tree modules) – сорс файловете на модула се намират в директорията на Линукс ядрото. Те са добавени в списъка за създаване на Makefile-овете на Линукс. След създаването, **стават част от обектовия код на ядрото и се стартират заедно с него**.

*__извън сорс-дървото__ на Линукс (out-of-tree modules) – сорс файловете са в отделна директория, която е свързана със сорс-дървото на Линукс чрез Makefile-а си. Когато се създаде модула, трябва да се копира в някоя системна директория на OS и да се стартира от скрипт, който се **изпълнява след като ядрото се е стартирало**.

Линукс за вградени системи

Видове драйвери от гледна точка на хардуера, който управляват:

*__платформени__ (platform) – устройства, които са в системата по време на стартирането ѝ и не се нуждаят от откриване – има дървесно двоично описание за тях. Пример – периферните модули на микроконтролер.

*__устройства__ (device) – устройства, които се включват в системата, след като тя е стартирала и се нуждаят от откриване. Пример – USB флаш памет.

*__магистрални__ (bus) – управляват магистралите и помагат за откриването на устройства.

Дървесни двоични описания

В персоналните компютри първата програма, която се изпълнява, след подаване на захранването, се казва BIOS (**B**asic **I**nput **O**utput **S**ystem).

Съхранява се във ROM/Flash чип.

Програмата е написана от производителя на дънната платка (motherboard), който знае точно какъв хардуер има на нея.

Дървесни двоични описания

Програмата инициализира, детектира и тества хардуера (**P**ower **O**n **S**elf **T**est, POST) на дънната платка, след което прехвърля управлението към стартиращата програма (bootloader) на операционната система [4].

При x86 архитектури, ресет векторът е на 0xFFFF.FFF0, където трябва да има инструкция за преход към BIOS програмата за конкретната дънна платка.

Когато OS стартира, тя работи без BIOS, освен когато използва функции за пестене на енергия (заспиване) – тогава използва ACPI функции на BIOS-а.

Дървесни двоични описания

UEFI (**U**nified **E**xtensible **F**irmware **I**nterface) – съвременната версия на BIOS, при която хардуерната конфигурация се запазва във .efi файл на отделен партишън на хард диска (EFI System Partition, ESP).

BIOS е ограничен до 1 MB, докато UEFI може да адреси 32- или 64-битово адресно поле.

Дървесни двоични описания

Вградените системи се проектират за конкретно приложение. Те не са универсални. Винаги целта при тях е да са максимално прости и затова нямат BIOS и UEFI.

Тогава – единственият начин е информацията за платформата да бъде “твърдо записана” (hard coded) в Линукс ядрото. При стартиране, bootloader-ът предава едно число на ядрото (machine type integer) и Линукс зарежда конфигурация, която отговаря на конкретния хардуер.

Дървесни двоични описания

Както всеки “твърдо записан” код, така и този води след себе си проблеми.

Платформите по цял свят се увеличават, особено след като *Линукс* започва да се използва във вградени системи[5], а програмистите, които поддържат ядрото (включително и Линус Торвалдс) са малко на брой.

Линус се ядосва на порт за демо платката Beagle Bone Black (TI Sitara AM335x), който включва много заплетени платформени файлове и решава, че е нужен нов подход [6].

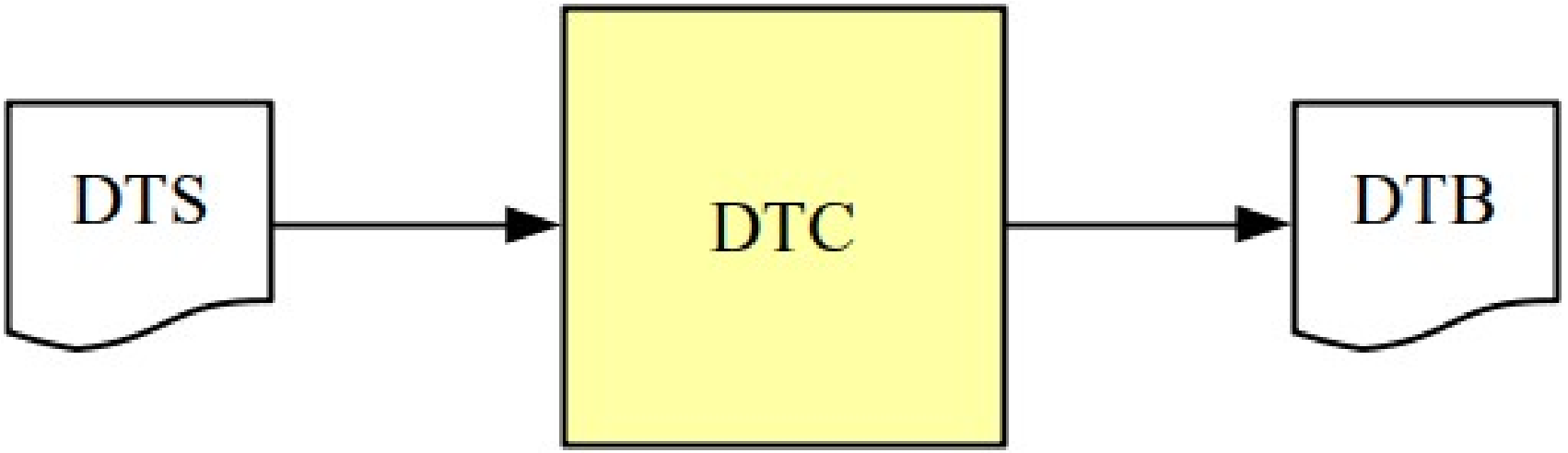
Дървесни двоични описания

Дървесни двоични описания (Device Tree Binary) – двоичен файл, който се чете от Линукс ядрото при стартиране и така се разбира кои софтуерни модули за кои хардуерни модули трябва да се заредят. Нещо повече – DTB съдържат и конфигурационни данни, например данни за мултиплексиране на изводите на микроконтролера, първоначални стойности, параметри на интерфейси и др.

Сорс код на дървесно двоично описание (**D**evice **T**ree **S**ource, DTS) – текстови файл, съдържащ описание на платформата (системата), от който се получава двоичното описание.

Дървесни двоични описания

Дървесен двоичен компилатор (Device **T**ree **C**ompiler, DTC) – програма, която преобразува сорс код на дървесно двоично описание в двоичен файл.



Дървесни двоични описания

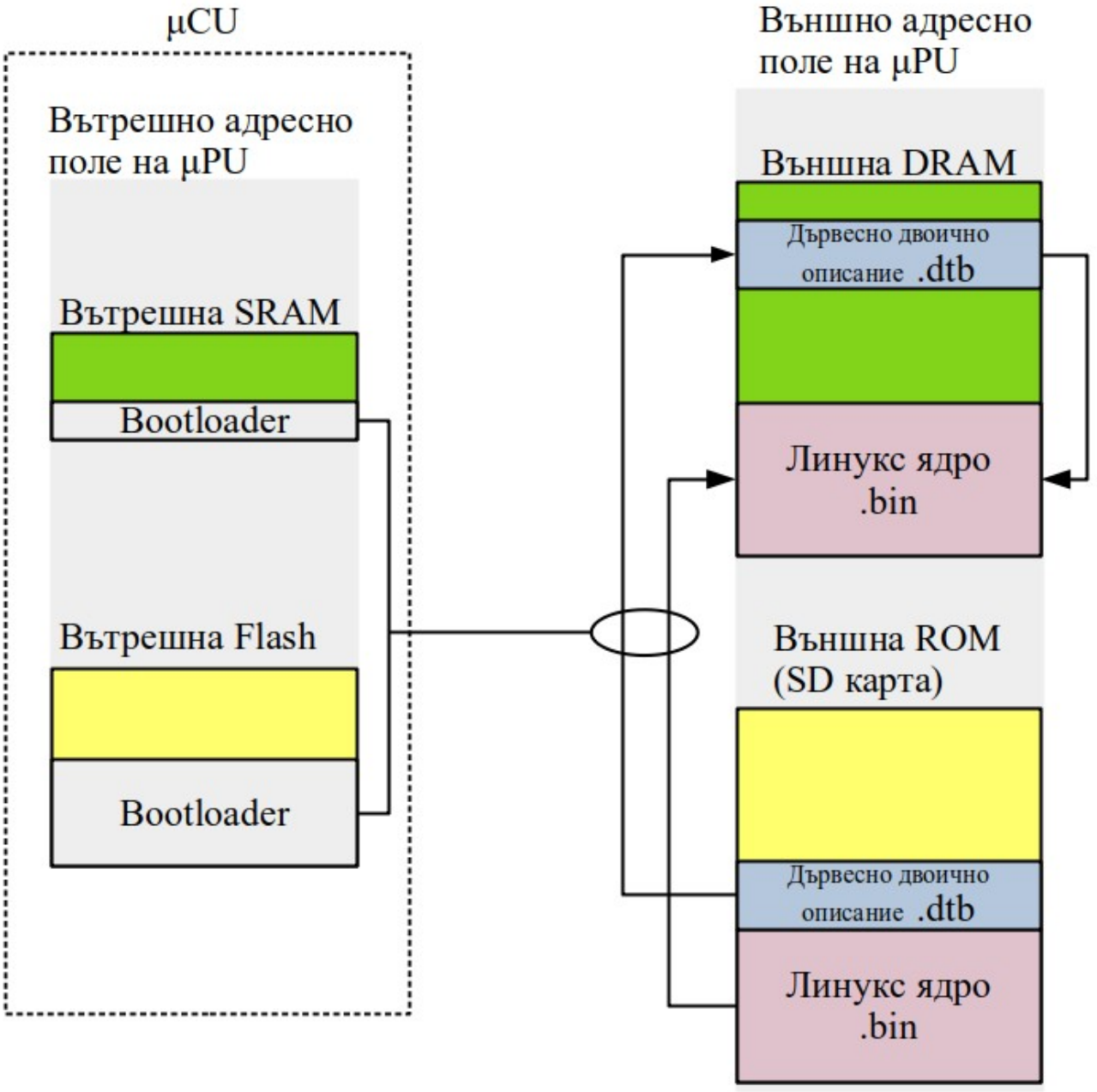
DTB се **зареждат предварително** от стартиращата програма (bootloader) на OS на определени адреси в RAM паметта, след което се предават като параметър на ядрото на OS.

Пример – команди, които U-boot изпълнява, за да стартира Линукс ядро с DTB:

```
mmc dev 0
fatload mmc 0 0xc0700000 /stm32f769-disco.dtb
fatload mmc 0 0xc0008000 /zImage

bootz 0x0c0008000 - 0xc0700000
```

Дървесни двоични описания



Дървесни двоични описания

Пример – дървесно двоично описание на QSPI флаш памет, внедрена в адресното поле на μPU (memory mapped). Драйверът се указва в “compatible” низа.

```
&qspi {
   pinctrl-0 = <&qspi_pins>;
   status = "okay";

   qflash0: n25q128a {
         #address-cells = <1>;
         #size-cells = <1>;
         compatible = "micron,n25q128a13", "spi-flash";
         spi-max-frequency = <108000000>;
         spi-tx-bus-width = <1>;
         spi-rx-bus-width = <1>;
         memory-map = <0x90000000 0x1000000>;
         reg = <0>;
   };
};
```

Литература

[1] Trevor Martin, „The Designer’s Guide to the Cortex-M Processor Family – A Tutorial Approach“, Elsevier, 2013.
[2] https://www.cs.columbia.edu/~hgs/os/sync.html
[3] D. Bovet, M. Cesati, “Understanding the Linux kernel”, O’Reilly, 2006.
[4] http://flint.cs.yale.edu/feng/cos/resources/BIOS/
[5] “Notes on the Beagle Bone Black”, technical note “About the Device Tree”, online, 2021.
https://www.ofitselfso.com/BeagleNotes/AboutTheDeviceTree.pdf
[6] P. Antoniou, “Board File to Device Tree Migration – A War Story”, online, 2021.
https://elinux.org/images/5/5c/ELCE2013_-_DT_War.pdf