

Операционни системи за реално време



Автор: гл. ас. 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!



Европейски социален фонд

Съдържание

1. Схеми за диспечериране на задачите (scheduling policies)
2. Комуникационни примитиви
3. Синхронизационни примитиви
4. Линукс за вградени системи
5. Дървесни двоични описания (device tree binaries)

Схеми за диспечериране на задачите

Операционна система за реално време (Real-Time Operating System, 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 (или още задачи, process, task) - програми, които се изпълняват в паралел от RTOS.

Нишки (threads) – 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

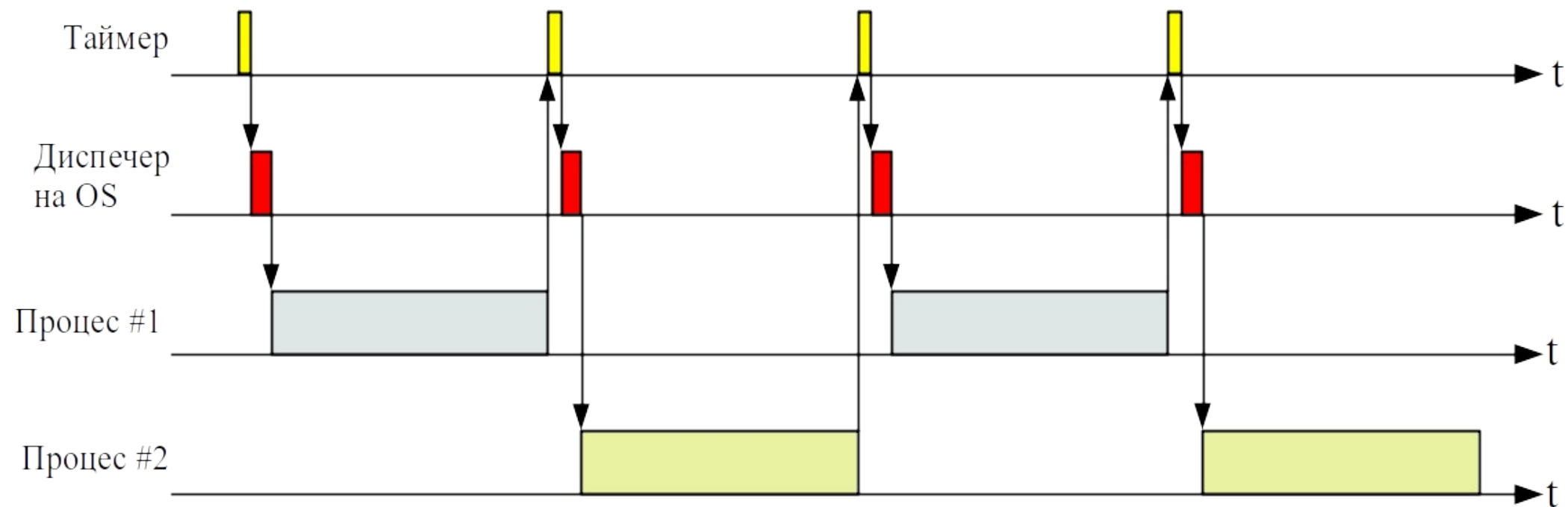
В микроконтролери без MMU понятието процес и нишка съвпадат.

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 μ PU ядро.

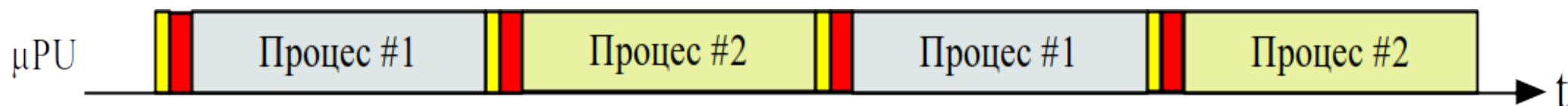
Квантовете от време се задават от един хардуерен таймер. Прекъсванията му прехвърлят процесорното време към диспечера.

Схеми за диспечериране на задачите

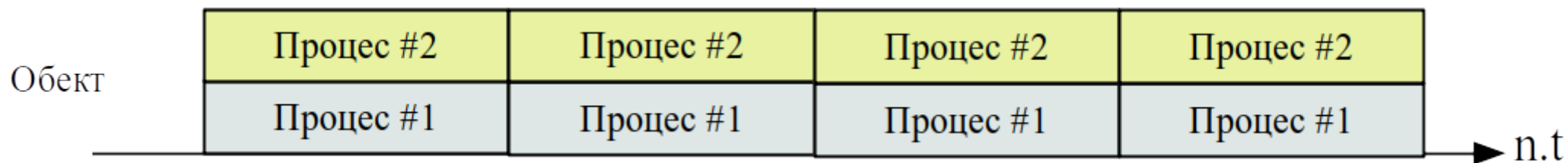


Схеми за диспечериране на задачите

Ето какво “вижда” μPU :



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



или с други думи, ако диспечерът превключва управлението на процесите 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

Отговорът е – зависи. **задачите**

Зависи от управлявания обект. Може да е:

* $x100$ ns

* $(x1 \div x100)$ μ s

* $(x1 \div x100)$ ms

* $(x1 \div x100)$ s

За голяма част от некритичните приложения $10 \div 100$ ms е достатъчно време.

Реално време – означава, че процесите 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 последици (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 последици (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

Процесите минават през три етапа по време на изпълнението си:

- ***блокирана** (wait) – процесът не се изпълнява от диспечера, защото чака някакво условие да се изпълни;
- ***изпълнява се** (run) – процесът не блокирал и се изпълнява от диспечера в неговия си времеви квант;
- ***готова за изпълнение** (ready) – процесът не е блокирал, но чака разрешение от диспечера, за да продължи да се изпълнява.

Схеми за диспечериране на задачите

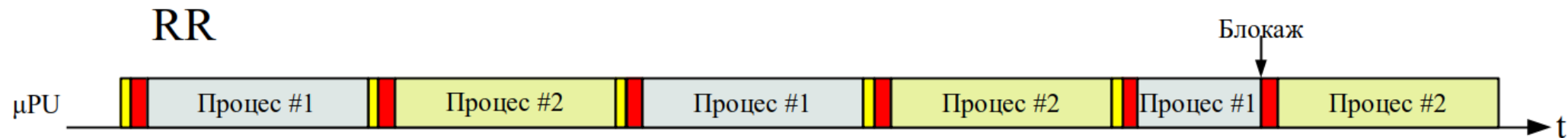
Схема за диспечериране на задачите (scheduling policy) – политика, която указва кога диспечера да даде микропроцесорно време на даден процес.

Най-често използваните схеми са:

- *round-robin (RR)
- *preemptive (PRE)
- *round-robin/preemptive (RR/PRE)
- *cooperative (COOP)

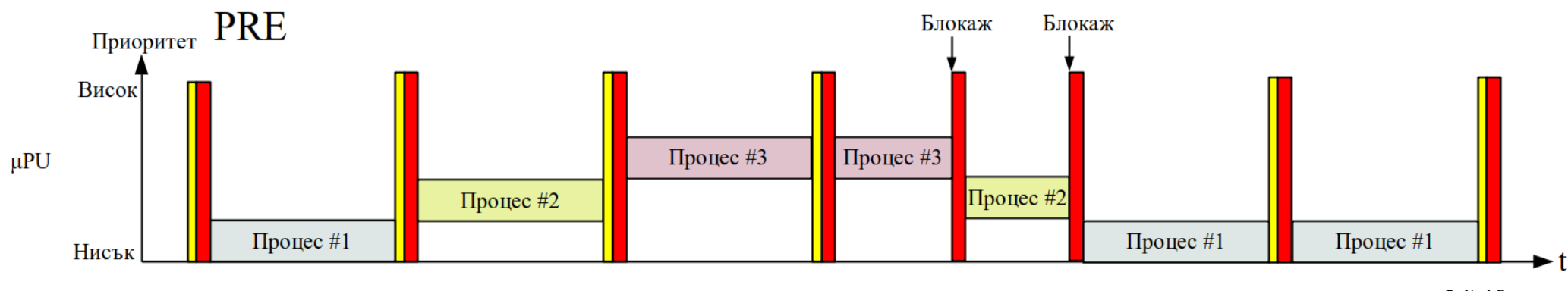
Схеми за диспечериране на задачите

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



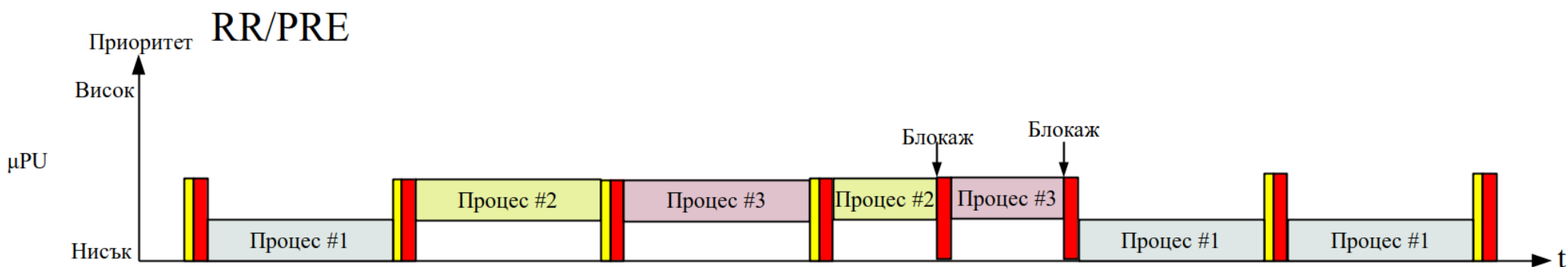
Схеми за диспечериране на задачите

Preemptive диспечериране (PRE) – на всеки процес се дава уникален приоритет. Не може да има два процеса с един и същи приоритет. Процеси с по-високи приоритети могат да прекъсват (preempt) процеси с по-ниски приоритети. Може да доведе до “приоритетен глад” (priority starvation), където процесите с ниски приоритети почти не се изпълняват заради процеси с по-високи приоритети.



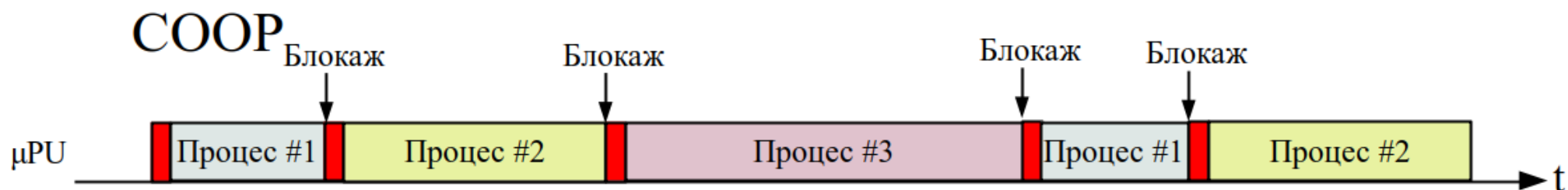
Схеми за диспечериране на задачите

Round-robin/Preemptive диспечериране (RR/PRE) – на всеки процес се дава приоритет. Може да има два процеса с един и същи приоритет. Процеси с различни приоритети се изпълняват по Preemptive схемата. Процесите с еднакви приоритети се изпълняват по RR схемата.



Схеми за диспечериране на задачите

Cooperative диспечериране (COOP) – всички процеси са с един и същ приоритет. Не се използва системен таймер, т.е. превключването от един процес в друг става без диспечер. Разчита се, че всеки един процес ще пуска следващия като се само блокира “доброволно”.



Комуникационни примитиви

Комуникационни примитиви (interthread communication) – променливи, които се използват за предаване на данни от един процес на друг под управлението на ядрото на RTOS.

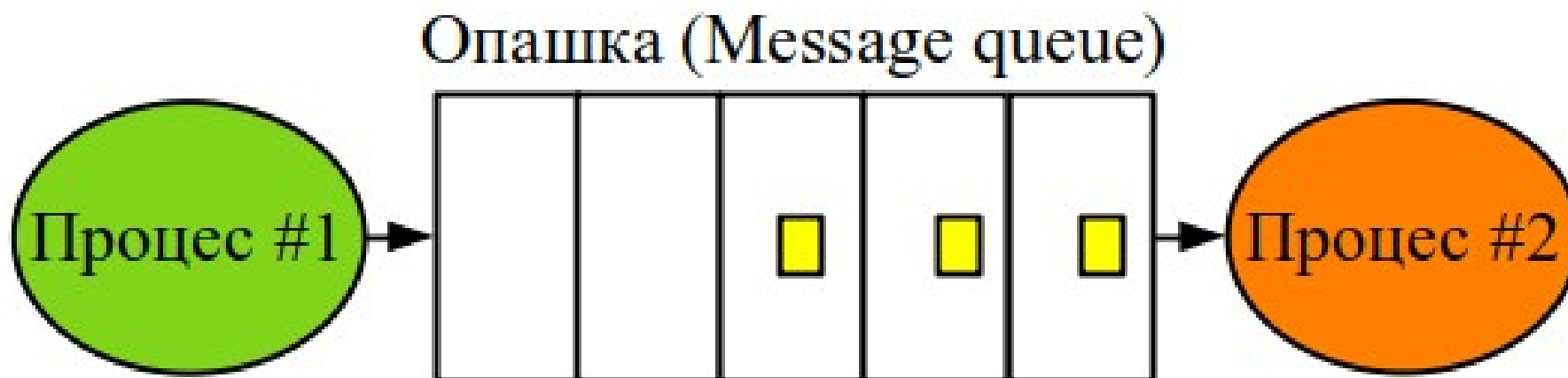
Най-често използваните ком. примитиви са:

- *опашки

- *поща

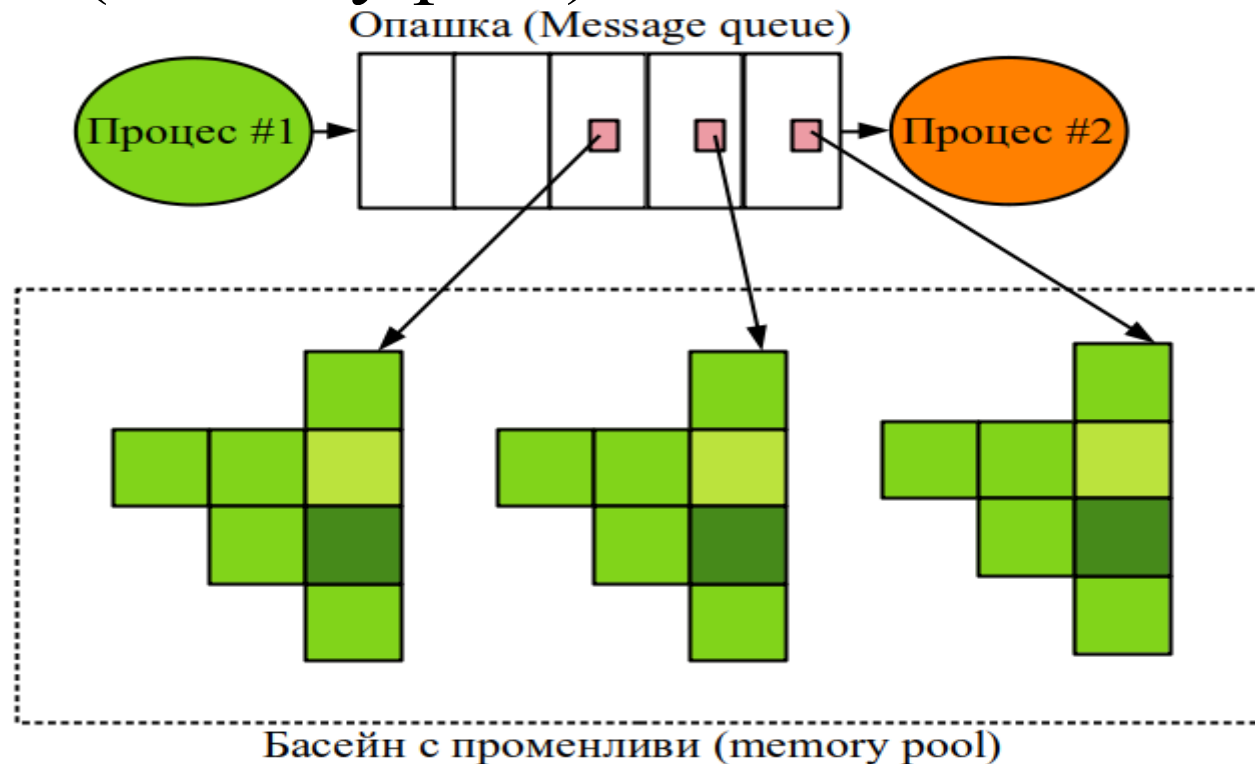
Комуникационни примитиви

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 (mail queue) – софтуерни FIFO буфери, които съдържат указатели към динамично заделени структури. Те се създават, четат и записват с API функции на ядрото на RTOS [1]. В този случай динамично заделената памет се нарича басейн с променливи (memory pool).



Синхронизационни примитиви

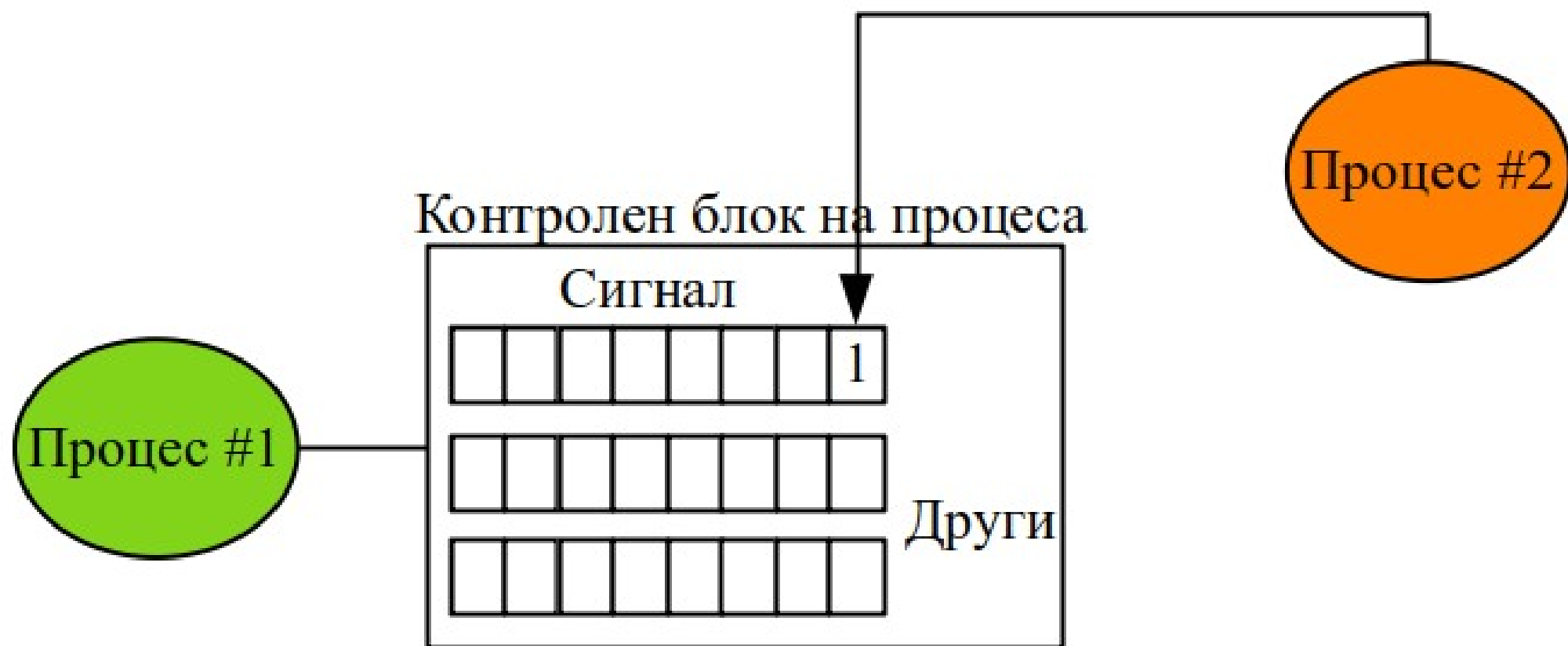
Синхронизационни примитиви (synchronization primitives) променливи, които се използват за синхронизиране изпълнението на един процес спрямо друг под управлението на ядрото на RTOS [1], [2].

Най-често се използват:

- *сигнали
- *семафори
- *мютекси (двоични семафори)
- *рандеву
- *бариера

Синхронизационни примитиви

Сигнал (signal) – променлива, която се намира в структурата, която описва процеса. Ако друг процес я достъпи, може да променя битовете на променливата и да я постави принудително в режим “блокиран” (wait).

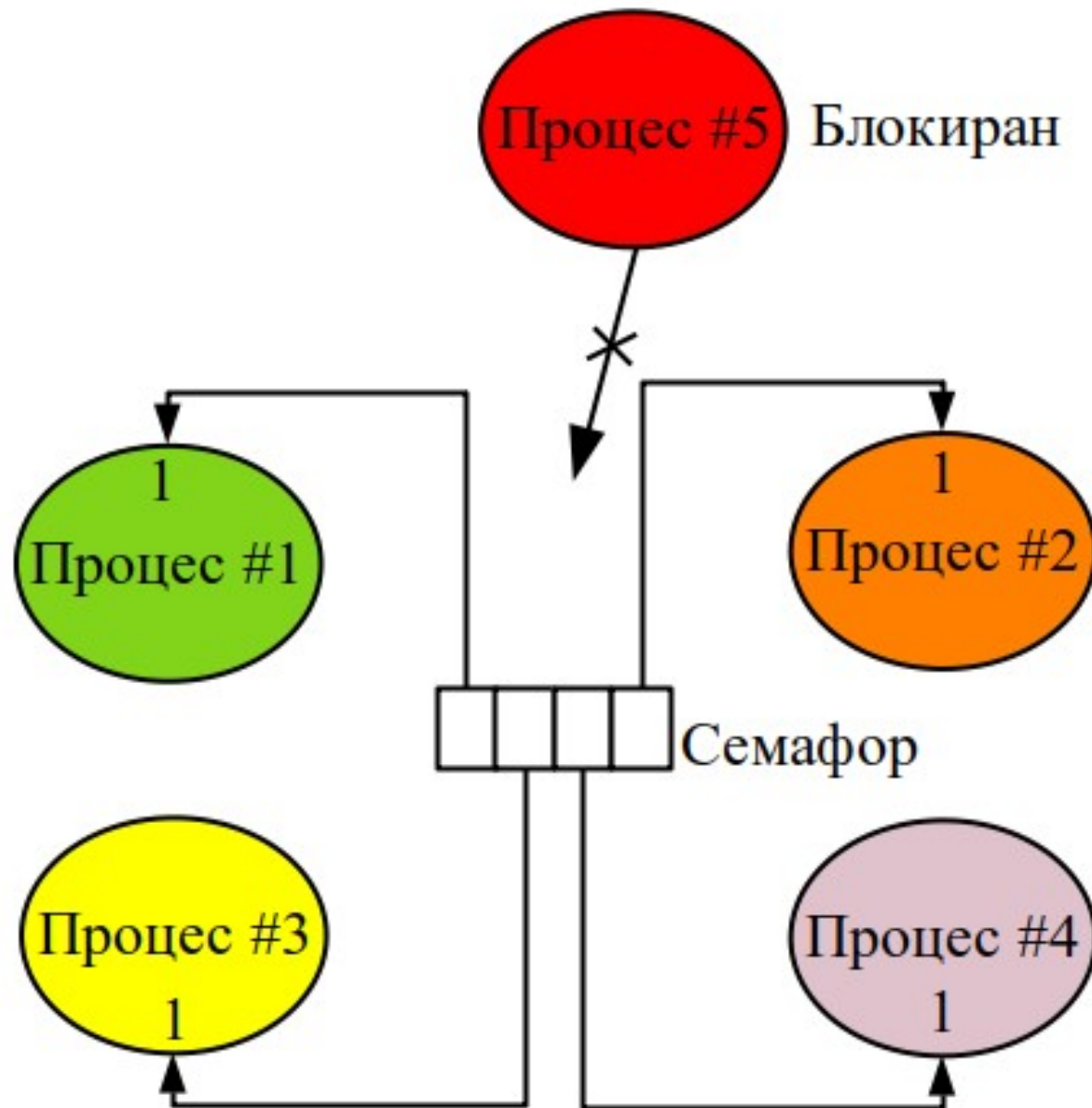


Синхронизационни примитиви

Семафор (semaphore) – променлива, която се зарежда с число от програмиста. Когато един процес се изпълнява, в тялото на кода му може да се извика функция, която достъпва тази променлива. Ако тя не е нула, процесът ще продължи напред. Ако тя е нула, процесът 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

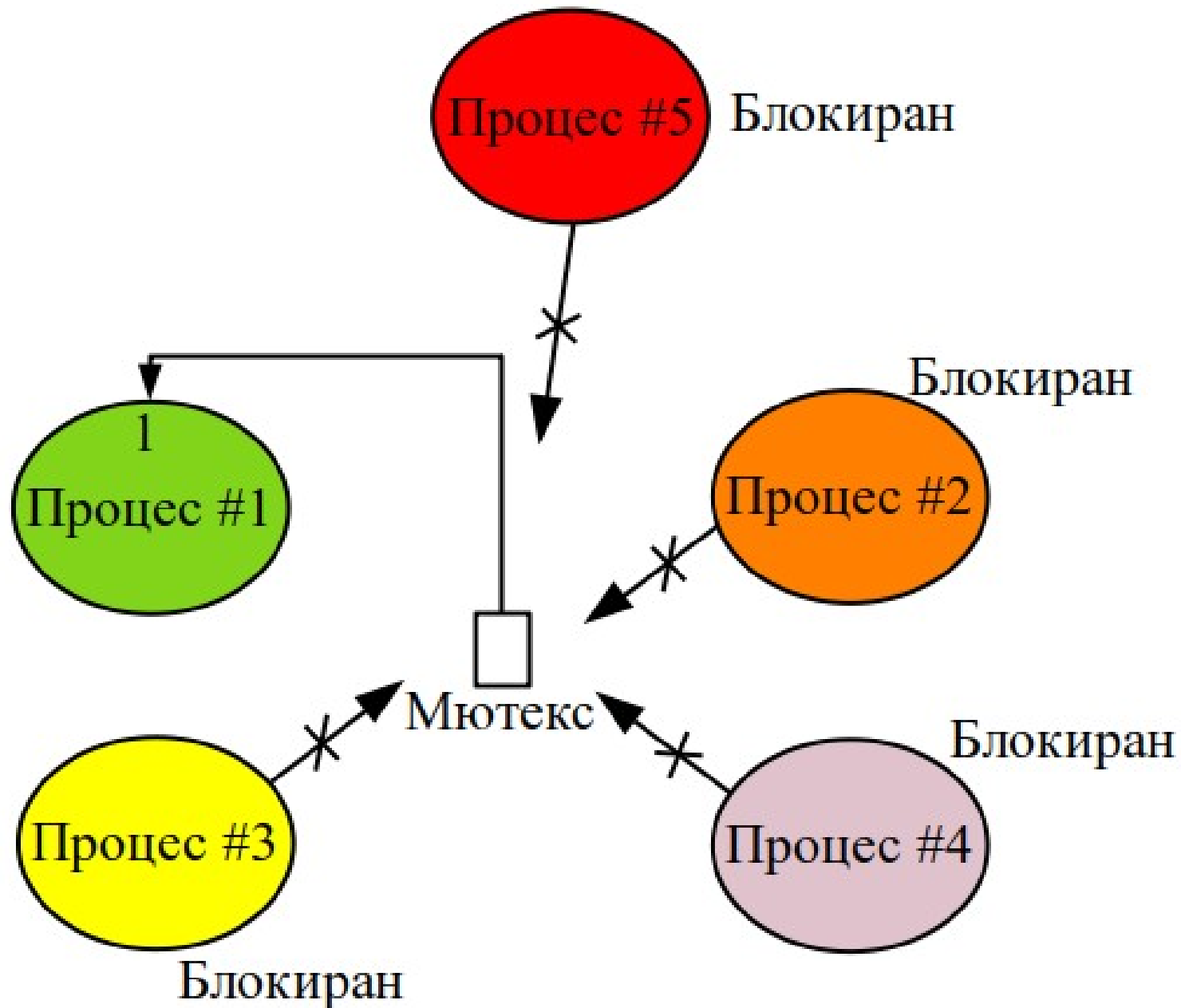


Синхронизационни примитиви

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 един процес, защото има само един сигнал TxD и изпраща данните серийно.

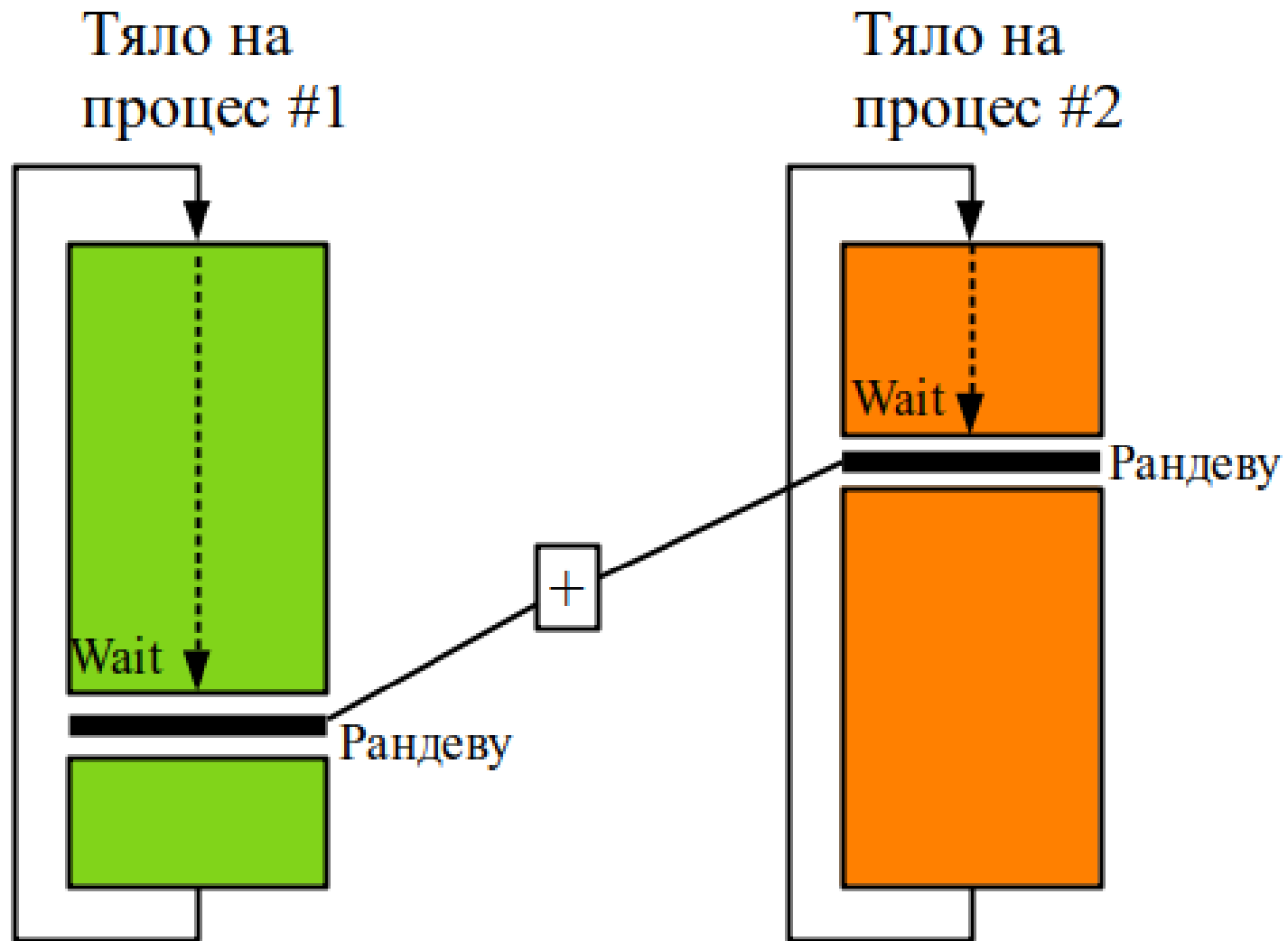
Синхронизационни примитиви



Синхронизационни примитиви

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

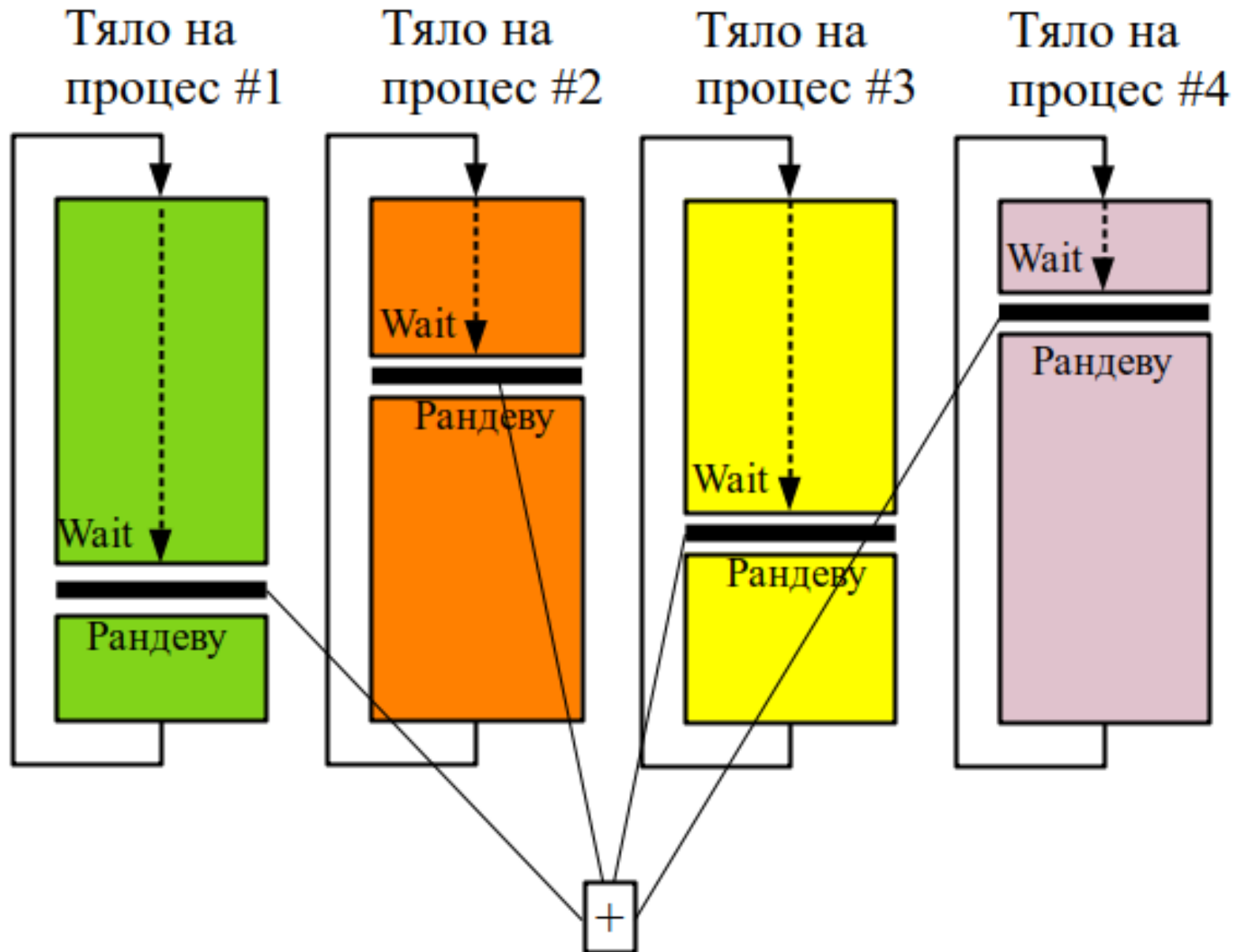
Синхронизационни примитиви



Синхронизационни примитиви

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

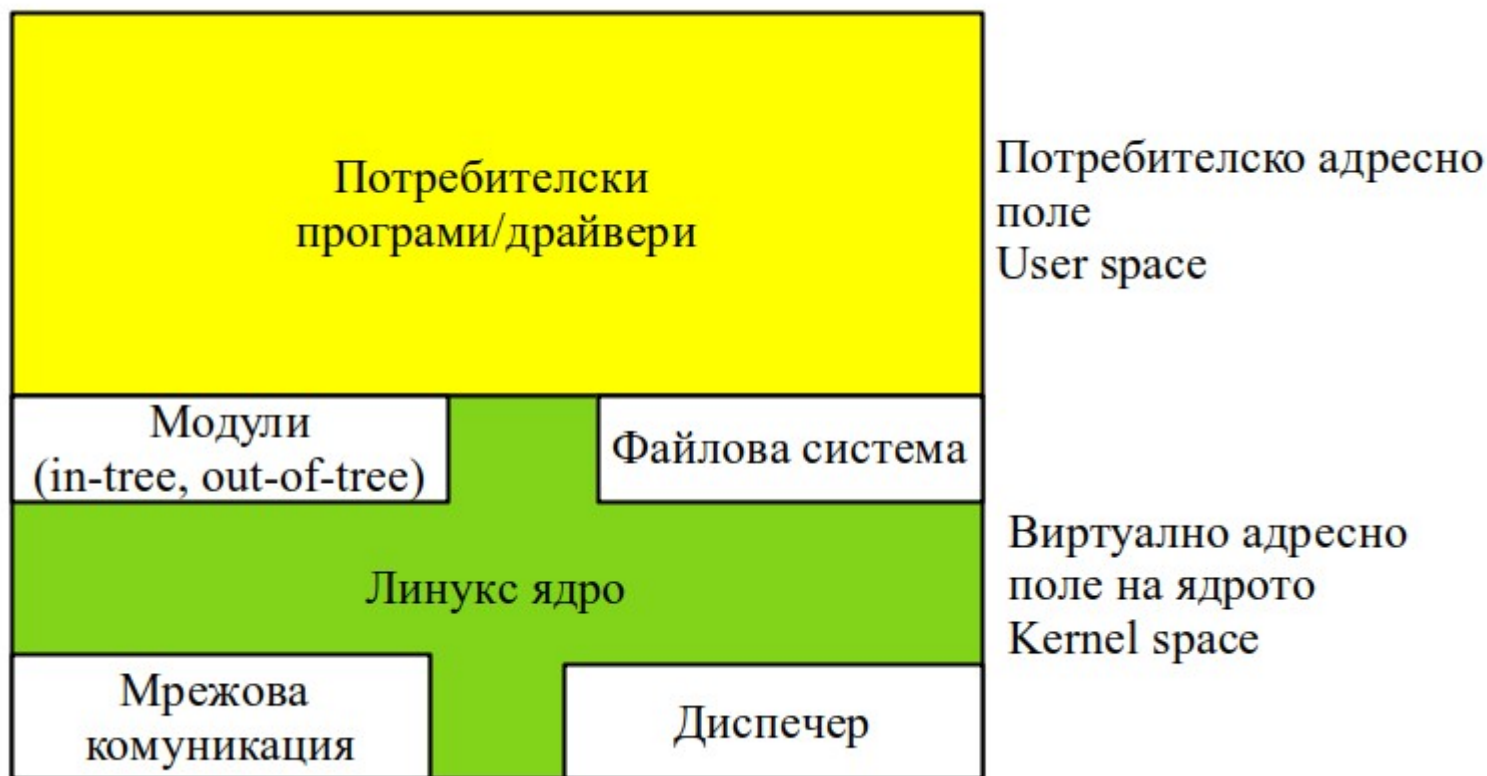
В по-голямата си част Линукс е **POSIX** съвместим, но не е POSIX сертифицирана (POSIX, IEEE 1003, е стандарт за операционни системи, дефиниращ формата на системните интерфейси, т.е. API функции, команден ред, поддържани команди, C библиотеки и др.).

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 **preemptive** 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 (**Power On Self Test, 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 (Unified Extensible Firmware Interface) — 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].

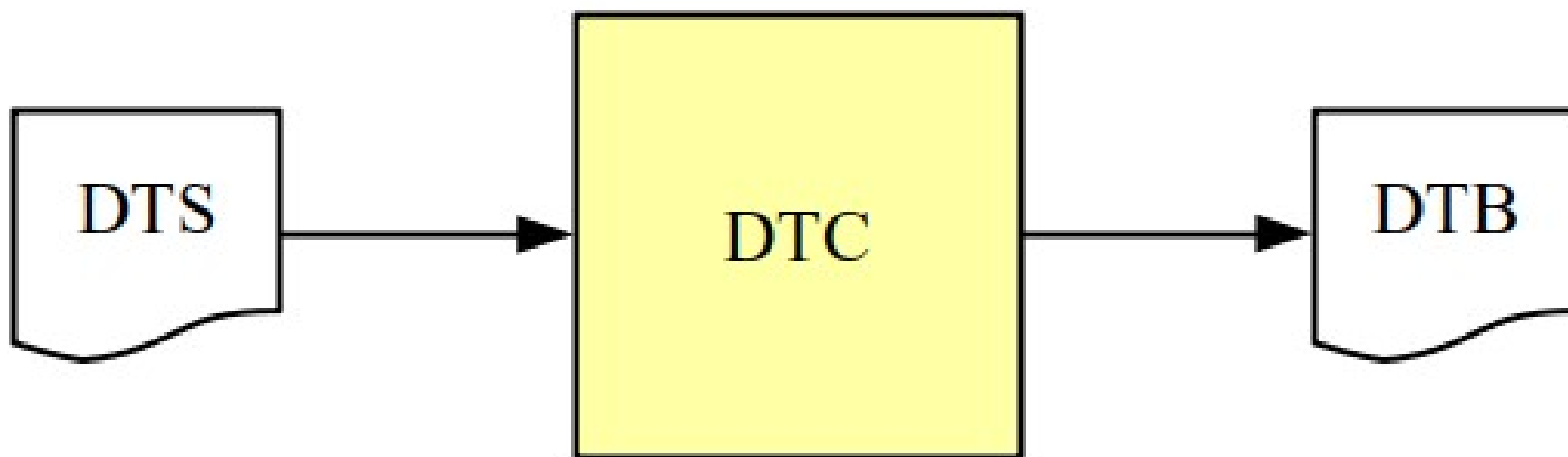
Дървесни двоични описания

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

Сорс код на дървесно двоично описание (Device Tree Source, DTS) – текстови файл, съдържащ описание на платформата (системата), от който се получава двоичното описание.

Дървесни двоични описания

Дървесен двоичен компилатор (Device Tree Compiler, DTC) – програма, която преобразува сорс код на дървесно двоично описание в двоичен файл.



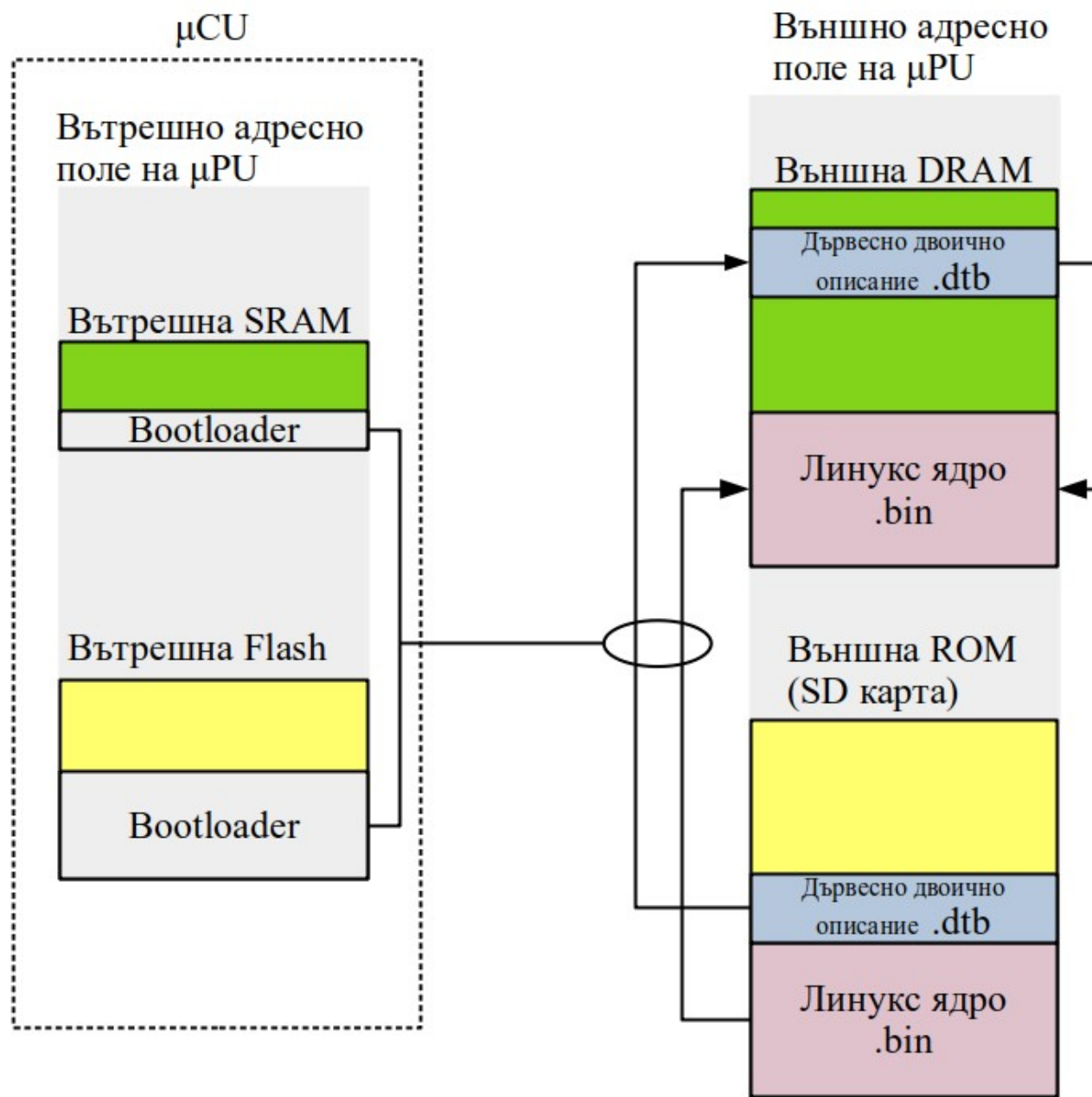
Дървесни двоични описания

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 0xc0008000 - 0xc0700000
```

Дървесни двоични описания



Дървесни двоични описания

Пример – дървесно двоично описание на QSPI флаш памет, внедрена в адресното поле на μ PU (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 0x10000000>;  
        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