

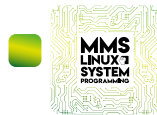
# Системно програмиране за Линукс

Мултипрограмиране и паралелни процеси. Взаимно изключване.

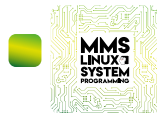
Ангел Чолаков



12.05.2021г.



This work is licensed under a Creative Commons  
“Attribution-ShareAlike 4.0 International” license.



# Съдържание I

- 1 Въведение
- 2 Многозадачност и мултипрограмиране
- 3 Програмни модели за многозадачна работа
- 4 Многозадачност чрез нишки
- 5 Състезание между процеси
- 6 Критична секция и взаимно изключване на процеси
- 7 Мъртва хватка
- 8 Подходи за осигуряване на взаимно изключване
- 9 Практическа част: запознаване с Posix Threads
- 10 Заключение



# Цел на презентацията

## ■ Да опита да:

- разшири представите ни за **многозадачност** и **паралелна обработка**;
- дефинира какво е **поток (нишка)** от изпълнявани инструкции;
- какви са предизвикателствата при обезпечаването на среда за изпълнение на множество процеси;
- опише как ОС подsigурява създаването и управлението на подпроцеси (нишки) в един процес;
- разясни какво са понятията **мъртва хватка** и **критична секция**;
- представи накратко основни алгоритми за **взаимно изключване** на процеси;



# Многозадачност и паралелизъм

## ■ Дефиниция за **паралелни задачи**:

- две или повече програмни приложения (процеса), изпълняващи се едновременно (или с времоделене) върху един или повече от един процесора или процесорни ядра;
- всяко едно от тези приложения би могло да включва в състава си един или повече потоци инструкции, диспечерирани от ОС



# Модел с множество процеси

## ■ Предпоставки:

- по-пълно оползотворяване на престояващите изчислителни блокове след инициране на входно-изходни операции, които са съпроводени със значителна латентност;
- необходимост от провеждане на самостоятелни по своя характер обработки, ангажиращи различни звена от системата (например: декодиране на мултимедиен поток от DSP и обмен на данни по Интернет);
- необходимост от ускоряване на изпълнението на програма чрез разделянето ѝ на припокриващи се подзадачи, които биха могли да се изпълнят едновременно;
- едновременно обслужване на заявки от множество потребители на системата

## ■ Системни и потребителски задачи:

- **системни** са тези, които са ангажирани с управлението на системните ресурси и са част от структурата на ОС;
- **потребителски**, които получават предварително зададени системни активи и се развиват под контрола на ОС



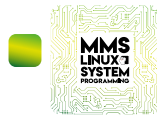
# Парарелни процеси

## ■ Разновидности:

- **независими** - функционират без пряко взаимодействие и споделяне на данни помежду си;
- **обвързани** - чието изпълнение и развитие във времето е свързано с ползване на общ ресурс, необходимост от синхронизация, комуникация и/или ползване на споделени данни

## ■ Типове свързани задания:

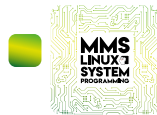
- **коопериращи се** - при които резултатът от работата на един процес оказва влияние върху функциите на други;
- **конкуриращи се** - при които има явно или индиректно съревнование за достъп до споделени апаратни и/или програмни средства



# Многозадачност и мултипрограмиране

## ■ Смесово разграничаване:

- исторически **многозадачността** се свързва с редуващото се във времето последователно превключване на процесора между няколко активни програми (чрез диспечериране с времеделене);
- **мултипроцесната** или **мултипрограмна** обработка - разпаралеляването на няколко задания се извършва върху система, снабдена с няколко процесора или процесорни ядра

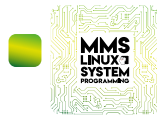




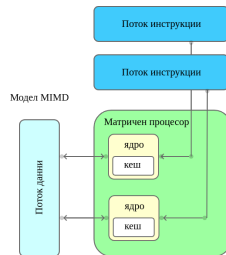
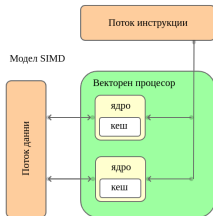
# Архитектури, обезпечаващи среда за мултипрограмиране

## ■ Примери:

- **SIMD** - единичен поток от еднотипни инструкции, приложени върху множество потоци от данни - т.нар. **векторни процесори**;
- **MIMD** - множество потоци от различни инструкции върху множество потоци от данни - т.нар. **мултипроцесори, мултикомпютри и клъстери** според организацията на паметта и топологията на свързване



# SIMD и MIMD: схематично представяне



pic. based on work by Vadikus, CC BY-SA 4.0 via Wikimedia Commons



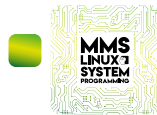
# Програмни модели за мултипрограмиране

## ■ Модел **клиент-сървър (client-server)**, състоящ се от:

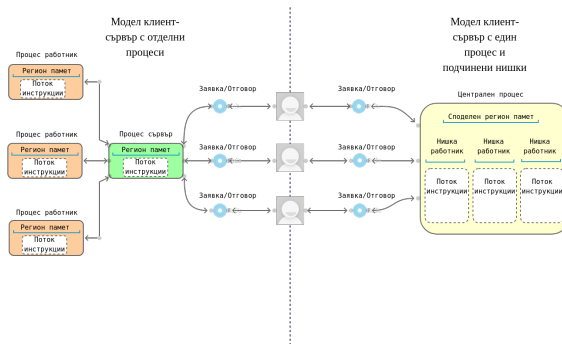
- компонент **сървър**, специализиран процес, отговорен за управлението на достъпа до споделен ресурс като процесор, файлова система или периферни устройства;
- един или множество **клиенти**, представлящи логически обекти, асоциирани с действителни потребители или техните процеси, които обслужват заявки за определена услуга;
- процесите, изобразяващи клиентските и сървърни инстанции, би могло да се развиват и самостоятелно в отделни логически адресни пространства

## ■ Модел **родител-работници (worker-threads)**:

- родителският процес и потомците се развиват паралелно и се кооперират, но споделят **общо адресно пространство**



# Програмни модели: илюстрация



pic. based on [https://en.wikipedia.org/wiki/Remote\\_procedure\\_call](https://en.wikipedia.org/wiki/Remote_procedure_call)



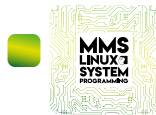
# Модел клиент-сървър

## ■ Недостатъци:

- централен сървър процес, който приема заявки и ги диспечерира посредством процеси-работници;
- всеки от новосъздадените процеси се разполага в отделна област на адресното пространство и консумира допълнителна памет и изчислителни ресурси;
- комуникацията с главния процес се усложнява в зависимост от модела на работа с паметта (сегменти споделена памет) и механизмите за синхронизация и обмен на съобщения



**Какъв подход се прилага за облекчаване част от описаните  
недостатъци?**



# Модел с потоци или нишки

## ■ Дефиниция:

- обособен **поток от инструкции**, поместен в границите на същото адресно пространство, което е присвоено и на процеса притежател;
- всеки процес притежава най-малко един основен поток от инструкции и нула или повече допълнителни потоци



# Многозадачност чрез нишки

## ■ Изпълняват се потоци от инструкции, чрез които:

- една задача може да се разпаралели на припокриващи се успоредно изпълнявани подзадания (вкл. провеждане на входно-изходни операции);
- породените нишки споделят общо адресно пространство с родителския процес (включително референции към глобални променливи, списък от налични експортирани функции и др.), което елиминира необходимостта от нови таблични структури и записи на странични съответствия;
- превключването на контекста се опростява, като се премахва изискването за зареждане на нова странична таблица при всяко превключване





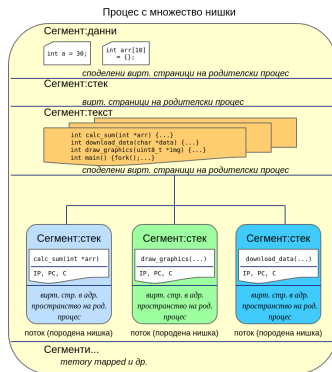
# Многопоточно изпълнение

## ■ Организацията на изобразяваните сегменти налага:

- представяне на всеки поток от инструкции чрез независим стеков сегмент с поддръжка на привързани регистрови структури: програмен брояч или указател на текущо изпълнявана инструкция (PC: program counter);
- спомагателни структури данни, представящи състоянието на всяка нишка и ползвани от ОС за управление (TCB - **Thread Control Block** - може да бъде аналогичен и на **Process Control Block** в някои ОС)



# Многопоточно изпълнение: карта на паметта



pic. based on <https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf>



# Предизвикателства в мултипрограмирането

## ■ За системните архитекти:

- проектиране на ОС, така че да се подsigури съвкупност от задачи с оптимално системно натоварване без прекомерно разходване на ресурси;
- оптимизиране на апаратните компоненти и постигане на ефикасен паралелизъм: както по отношение на достъпваните данни, така и по отношение на изпълняваните инструкции;
- разработване на механизми за надеждна междупроцесна комуникация и синхронизация при работа с множество процеси и потоци;
- предоставяне на усъвършенствани програмни средства за профилиране и отстраняване на дефекти



# Предизвикателства в мултипрограмирането: продължение

## ■ За софтуерните разработчици:

- идентифициране на елементи от едно задание, които подлежат на разпаралеляване;
- разрешаване на зависимости между подзадачите и гарантиране интегритет на обработваните данни;
- постигане на безпроблемна програмна синхронизация между коопериращите се и съревноваващи се процеси или нишки;



# Многопоточен програмен модел: реализация

## ■ Модел **много-към-едно (many-to-one)**:

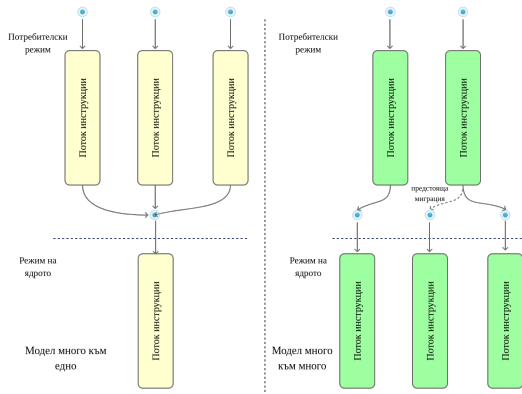
- множество нишки в рамките на един процес са привързани към една системна нишка в контекста на ядрото, като последната подsigурява изпълнението на заявените системни извиквания от страна на ОС;
- управлението на многозадачната работа в потребителското пространство е поверено на системна библиотека, но обръщението към една нишка в ОС прави невъзможно действително хардуерно разпаралеляване;
- всяко блокиращо системно извикване блокира и останалите логически нишки

## ■ Модел **едно-към-едно (one-to-one)** и **много-към-много (many-to-many)**:

- за всеки потребителски поток ОС създава и привързва отделна системна нишка на ядрото, която е входна точка за обработване на системни извиквания;
- механизмът на съответствие може да е статичен или да повери на диспечера на процеси да преразпределя асоциациите



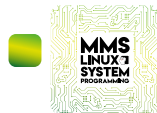
# Илюстрация на описаните модели



pic. based on [https://en.wikipedia.org/wiki/Thread\\_\(computing\)](https://en.wikipedia.org/wiki/Thread_(computing))



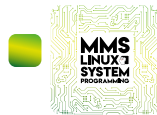
**Какъв е подходът, ползван в Линукс? Има ли ограничение за  
максималния брой създавани нишки?**



# Фактори, усложняващи комуникацията и синхронизацията

## ■ Открояват се:

- диспечерът на ОС е в състояние да прекъсне и временно преустанови изпълняван процес или поток по всяко време;
- достъпът до споделени данни или сегменти памет създава предпоставки за нежелани изменения в среда с множество потоци;
- множество коопериращи или конкуриращи се процеси или потоци се съревновават за изпълнение и ползване на системни ресурси





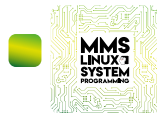
# Какво е състезание между процеси или потоци?

## ■ Дефиниция:

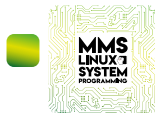
- ситуация, която възниква при неопределено във времето изпълнение на множество потоци от инструкции и при която крайният резултат от обработките не е предвидим и дефиниран

## ■ Пример:

- един процес или нишка извършва операции по четене на споделени данни, а в същия този момент друг процес или нишка прави изменение на съдържанието на общодостъпните референции данни без наличие на синхрон



**Защо е възможна появата на неочаквано или нежелано състезание?  
Не са ли всички инструкции апаратно синхронизирани? Достъпът и  
обновяването на паметта не настъпват ли мигновено?**



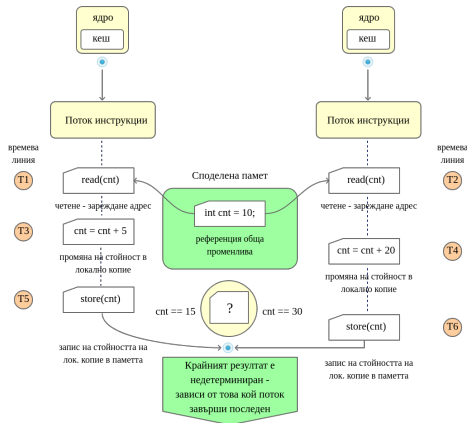
# Възникване на междупроцесни състезания

## ■ Предпоставки:

- асинхронна природа на диспечериране и хардуерно обслужване на програми, обвързано с разпределяне на процесора или присвояването ядра;
- конвейерна природа на извличане, изпълнение на инструкции и вътрешната им имплементация в рамките на дадена микроархитектура;
- наличие на вътрешно буфериране на стойности и управление на йерархични каталожни структури в процеса на виртуализация на паметта;
- странично-сегментна организация на паметта с възможност за обособяване на споделени сегменти и реферирането им от различни потоци



# Състезание на процеси: диаграма

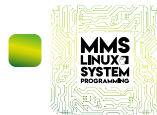


pic. based on [https://en.wikipedia.org/wiki/Race\\_condition](https://en.wikipedia.org/wiki/Race_condition)



# Междупроцесно съревнование: пример

- Коопериращи се процеси или потоци, като:
  - единият прочита съдържанието на общодостъпна целочислена променлива и го увеличава с константа;
  - вторият инициира аналогично четене, но намалява съдържанието на променливата с различна константа;
- Крайният резултат зависи от:
  - действителния ред на изпълнение на операциите във времето;
  - представянето на тези операции в последователности от регистрови обработки, като не се гарантира, че една логическа програмна инструкция ще съответства на и ще се реализира физически от една процесорна инструкция



# Пояснение към примера

- Нека се фокусираме върху прочитането на споделена променлива:

- `int a = 20;`

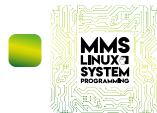
- `int res = a + 10;`

- Изчислението би могло да се представи чрез следната поредица инструкции:

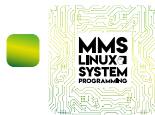
- `mov 0xffff9a1c, %eax`

- `add $0xA, %eax`

- `mov %eax, 0xffff9a1c`



**Време за демонстрация с опростена тестова програма...**



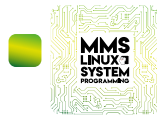
# Особености при модел със състезание

## ■ Резултат при състезание:

- резултатът от обработката е **недетерминиран** и зависи от действителното асинхронно развитие на потоците във времето

## ■ Недетерминираността се дължи на:

- характера на избор на активна задача от страна на системния диспечер и неговият алгоритъм на работа;
- неопределените моменти на изпълнение на системните извиквания и съпътстващите ги превключвания на контекста;
- статистически параметри на цялостния програмен товар, обслужван от системата във времето





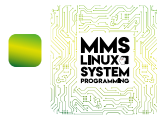
# Какво провокира програмно състезание?

## ■ Същност:

- наличие на последователност от обработки, принадлежащи на повече от една програмни нишки, които достъпват конкурентно споделен ресурс (променлива или обща памет в описания пример)
- регионът от кода, обвързан с манипулиране на споделения ресурс, се нарича **критична секция**



## Как да се предпазим от нежелани съревнования?



# Критична секция и взаимно изключване

## ■ Същност:

- защитен срещу едновременно паралелно изпълнение регион от инструкции с операции по четене и/или промяна на споделени данни;
- оригиналната дефиниция на това понятие се приписва на Edsger Dijkstra (с референция към статията: "Cooperating Sequential Processes", 1968, <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html>)



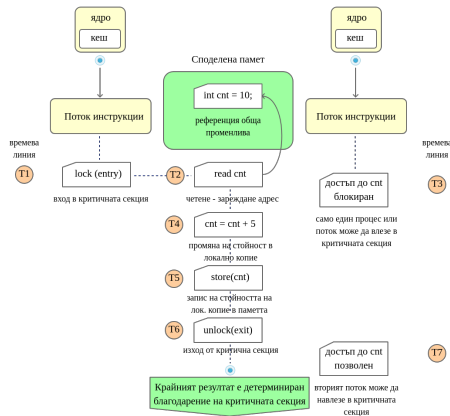
# Техника на взаимно изключване на процеси

## ■ Условия за **взаимното изключване (mutual exclusion)**:

- 1 само един процес може да се намира в критичната секция след като е получил достъп, а останалите са задължени да изчакат;
- 2 допуска се, че оторизираният процес може да престои в критичната секция крайно време;
- 3 предполага се, че състезаващите се за достъп потоци също биха получили възможност да влязат в защитената секция за крайно време, но само чрез редуване;
- 4 изборът на следващ допускан процес се прави само измежду престояващите в очакване, които не изпълняват в момента код, принадлежащ на критичната секция



# Взаимно изключване: илюстрация

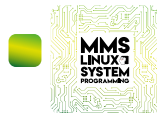


pic. based on [https://en.wikipedia.org/wiki/Mutual\\_exclusion](https://en.wikipedia.org/wiki/Mutual_exclusion)



# Потенциални проблеми за разрешаване

- При нарушение на описаните условия би могло да възникне:
  - при нарушение на 2. - взаимно блокиране, известно като **мъртва хватка (deadlock)**;
  - при неистинност на 3. - **безкрайно отлагане**



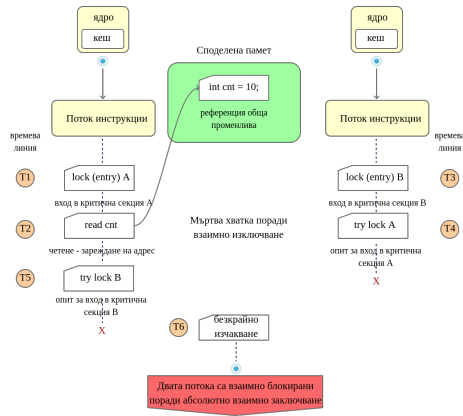
# Възникване на мъртва хватка

## ■ Възможни причини:

- взаимно блокиране между процеси чрез последователно заемане на блокиращ ресурс, който никой не освобождава;
- некооперативен процес, който се отлага безкрайно и не позволява други да навлезат в заета критична секция;



# Мъртва хватка: илюстрация



pic. based on <https://en.wikipedia.org/wiki/Deadlock>





# Отговорности на ОС при мултипрограмиране

## ■ Включват:

- предоставяне на средства (**синхронизационни примитиви**) за групиране на блок операции в критична секция или атомарна защитена транзакция;
- снабдяване с механизми за целесъобразна междупроцесна комуникация и координация;
- използване на апаратни и програмни средства за управление на конкуриращи се паралелни процеси и потоци



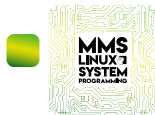
# Възможни подходи за взаимно изключване

## ■ Възможности:

- подход с **монополизиране на процесора** и предотвратяване на паралелно изпълнение на блок от операции върху други ядра или процесори в рамките на критичната секция (атомарни операции и ползване на специализирани инструкции);
- подход с **блокировка и арбитраж на паметта** в рамките на критичната секция, реализирани отново чрез специфична апаратна поддръжка и набор от инструкции;



**А какви алгоритми за взаимно изключване съществуват?**



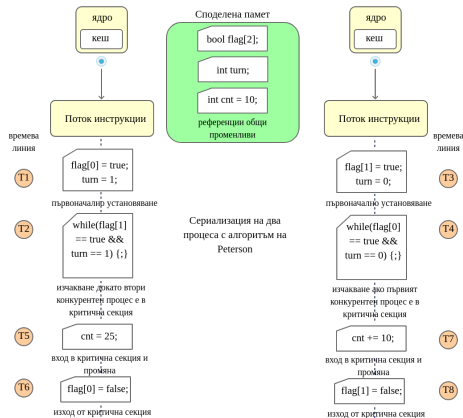
# Взаимно изключване на два процеса

## ■ Алгоритъм на Peterson:

- разработен от Gary Lynn Peterson, изследовател и учен в областта на компютърните науки;
- ползва масив от Булеви флагове, обозначаващи готовност за всеки от двата конкуриращи се потока;
- целочислена променлива, която пази реда на следващия позволен за изпълнение процес;
- поток, желаещ да навлезе в критична секция, установява своя Булев флаг на истинна стойност и записва в реда индекса на конкурента;
- след изпълнение на обработката и изход от критичната секция всеки процес съхранява в своя Булев флаг стойност неистина;
- така се подsigурява, че всеки от чакащите процеси ще получи шанс да манипулира ексклузивно споделените данни



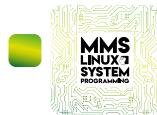
# Алгоритъм на Peterson: диаграма



pic. based on [https://en.wikipedia.org/wiki/Peterson%27s\\_algorithm](https://en.wikipedia.org/wiki/Peterson%27s_algorithm)



## Време за демонстрация: пример с имплементация на алогиритъма на Peterson



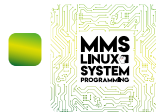
# Взаимно изключване на $n$ процеса

## ■ Bakery алгоритъм:

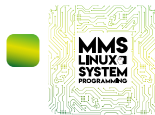
- развит от Leslie Lamport, разработчик на  $\text{\LaTeX}$  и изследовател в областта на разпределените системи;
- всеки от конкуриращите се процеси получава номер в нарастваща последователност при заявка за достъп до споделен ресурс;
- ако два процеса получат еднакъв номер, приоритет се дава на потока с по-малък системен идентификатор;
- с приоритет в критичната секция навлиза потокът с най-малък разпределен номер или съобразно подхода за разрешаване на конфликт;
- предвид нарастващите почти уникални идентификатори, алгоритъмът е детерминиран

## ■ Повече информация:

- <https://amturing.acm.org/pdf/LamportTuringTranscript%20.pdf>



## Практическа част: представяне на потоците в Линукс, примери с Posix Threads (libpthread).





# Разграничение между процеси и потоци в Линукс

## ■ Имплементация:

- в Линукс ОС се прилага модел на съответствие **едно-към-едно** в изобразяването на потребителски процес в процес на ниво ядро;
- не се прави съществена разлика в третирането на потоци и процеси;
- всеки поток се третира като **LWP (Light Weight Process)** олекотен процес, като потоците също се представят с помощта на процеси в ядрото;
- една потребителска програма е в състояние да поражда и развива множество потоци, като в този случай асоциираните процеси в ядрото споделят един и същи **групов идентификатор (TGID - thread group identifier)**



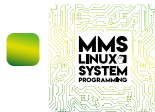
# Състояния на процесите в Линукс

## ■ Пояснение:

- **Sleeping/Waiting** - (в очакване) или спящи с две разновидности: (interruptible) очакващи програмно прекъсване посредством сигнал или (uninterruptible), които са ангажирани с провеждане на входно-изходна операция или в очакване на апаратно прекъсване;
- **Running** - (в готовност) като в тази категория попадат готовите за изпълнение и изпълняващите се задачи;
- **Stopped** - (преустановен), приведен в това състояние посредством сигнал SIGSTOP и по-късно възобновен чрез обработване на сигнал SIGCONT;
- **Zombie** - (зомби), процес, който е привършил работа, но поради нарушена верига на комуникация или взаимодействие с други процеси неговият контролен блок за управление все още се намира в структурите данни на ОС



## Какви програмни средства съществуват за разработка на многопоточкови програми?



# Въведение в Posix threads

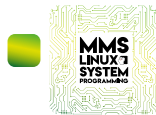
## ■ Защо Posix threads и pthread библиотека?

- разрешава проблема с грижа за особеностите и различията, свързани с апаратната поддръжка на процеси и потоци на ниво микроархитектура;
- въвежда стандартизиран програмен интерфейс и набор от API функции за имплементиране на многопоточна функционалност;
- предоставя ефикасна среда за олекотено създаване на нишки, които споделят общо адресно пространство с основния пораждащ процес без утежнения при работа със споделена памет;
- освобождава разработчиците от познаване на спецификите и детайлите на поддържаните от ОС системни извиквания за многопоточна работа;
- за UNIX подобните и съвместими ОС Pthreads интерфейсет е стандартизиран като част от [https://standards.ieee.org/standard/1003\\_1-2008.html](https://standards.ieee.org/standard/1003_1-2008.html)



# Pthreads: категории налични функции

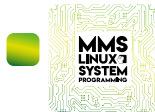
- Приложните функции са групирани в четири класа:
  - **управление на потоци**: създаване, обединение, управление на атрибути и др.;
  - **синхронизация на потоци**: посредством специализирани помощни примитиви като **ключалки и бариери (locks and barriers)**;
  - **взаимно изключване**: чрез употреба на примитиви за обособяване на критични секции (**mutex**);
  - **управление на условни блокове и нотификация**: с помощта на функции за боравене с **условни променливи (condition variables)**



# Pthreads: насоки при разработка

## ■ Именуване на функциите:

- конвенцията приема употреба на префикс: `pthread_`;
- процедурите по компилиране и свързване налагат явно указване на Pthreads библиотеката (`-lpthread` за `gcc`);
- повече инфо може да откриете тук: <https://hpc-tutorials.llnl.gov/posix/compiling/>



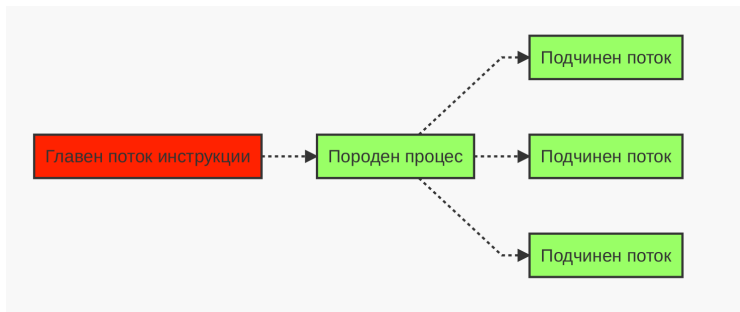
# Pthreads: създаване и управление на нишки

## ■ Важни прототипи на функции:

- `pthread_attr_init ( attr )` - инициализиране на обект с атрибути, включващи статут, политика на диспечериране, размер на стека и др.;
- `pthread_attr_destroy ( attr )` - освобождаване на референцията към атрибутите;
- `pthread_create ( thread , attr , start_routine , arg )` - създаване на нишка и запускането ѝ чрез изпълнение на указаната функция: `start_routine`;
- `pthread_cancel ( thread )` - заявка за терминиране на изпълнението;
- `pthread_exit ( status )` - терминиране на текущо изпълняваната нишка



# Pthreads: процес с няколко нишки - илюстрация



pic. based on [https://hpc-tutorials.llnl.gov/posix/creating\\_and\\_terminating/](https://hpc-tutorials.llnl.gov/posix/creating_and_terminating/)





# Pthreads: диспечериране на потоци

## ■ Особенности:

- Pthread API не предоставя механизми за обвързване изпълнението на поток с конкретно ядро или процесор;
- съществуват непреносими решения като `pthread_setaffinity_np` ;
- функции, за чиято имплементация е отговорна Линукс ОС, включват например: `sched_getaffinity` и `sched_setaffinity`



# Проектиране на паралелни задания

## ■ Съборажения:

- избор на удачен и ефективен програмен модел и средства за поддръжка;
- удачно разделяне на задачата на независими или подаващи се на оптимизация подзадачи;
- стратегия за балансиране на работния товар
- избор на подходящи механизми за междупроцесна комуникация и синхронизация;
- арбитражиране на достъпа до паметта;
- оценка на вложените усилия и разходи за разработка



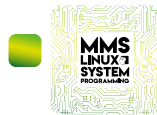
# Обезпечаване на паралелизъм

## ■ Обобщение на програмни модели:

- **Manager/Workers** - една основна управляваща нишка (Manager), която приема входни данни и разпределя координираната им обработка от няколко подпроцеса (Workers);
- **Pipeline (конвейер)** - пример за разделяне на заданието на последователни подоперации, всяка от които се изпълнява независимо от отделен поток обработчик;
- **Peer (кооперативен)** - подобен на Manager/Worker, но в този случай Manager потокът също участва и в пряката обработка на данни



**Предстои да разгледаме някои механизми и примитиви за  
междупроцесна синхронизация и комуникация в следващите лекции**



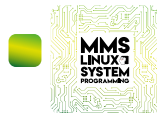
# Бележки по материалите и изложението

- материалът е изготвен с образователна цел;
- съставителите не носят отговорност относно употребата и евентуални последствия;
- съставителите се стремят да използват публично достъпни източници на информация и разчитат на достоверността и статута на прилаганите или реферирани материали;
- текстът може да съдържа наименования на корпорации, продукти и/или графични изображения (изобразяващи продукти), които може да са търговска марка или предмет на авторско право - ексклузивна собственост на съотнесените лица;
- референциите могат да бъдат обект на други лицензи и лицензни ограничения;
- съставителите не претендират за пълнота, определено ниво на качество и конкретна пригодност на изложението;
- съставителите не носят отговорност и за допуснати фактологически или други неточности;
- свободни сте да създавате и разпространявате копия съгласно посочения лиценз;



# Референции към полезни източници на информация

- <https://en.wikipedia.org/>
- [https://en.wikipedia.org/wiki/Multithreading\\_\(computer\\_architecture\)](https://en.wikipedia.org/wiki/Multithreading_(computer_architecture))
- [https://en.wikipedia.org/wiki/Peterson%27s\\_algorithm](https://en.wikipedia.org/wiki/Peterson%27s_algorithm)
- [https://en.wikipedia.org/wiki/Lamport%27s\\_bakery\\_algorithm](https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm)
- <https://en.wikipedia.org/wiki/Multiprocessing>
- <https://en.wikipedia.org/wiki/SIMD>
- <https://en.wikipedia.org/wiki/MIMD>
- [https://en.wikipedia.org/wiki/Critical\\_section](https://en.wikipedia.org/wiki/Critical_section)
- <https://en.wikipedia.org/wiki/Deadlock>
- [https://man7.org/linux/man-pages/man3/pthread\\_setaffinity\\_np.3.html](https://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html)
- <https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html>
- <https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf>
- [https://hpc-tutorials.llnl.gov/posix/creating\\_and\\_terminating/](https://hpc-tutorials.llnl.gov/posix/creating_and_terminating/)
- <https://search.creativecommons.org/>



**Благодаря Ви за вниманието!**

