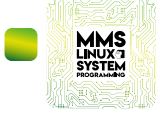
Системно програмиране за Линукс

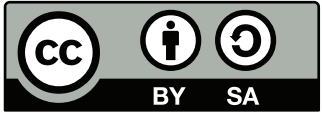
Средства и механизми за междупроцесна синхронизация.

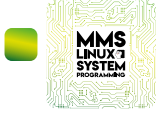
Ангел Чолаков

MMSolutions
A ThunderSoft Company

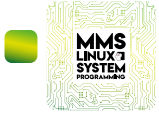
19.05.2021г.





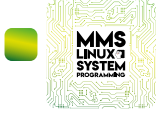


Съдържание I

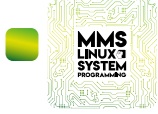


Цел на презентацията

- Да опита да:
  - поясни защо се налага синхронизация на задачи и потоци;
  - посочи често прилагани способи за реализиране на критични секции;
  - разясни какви средства предоставя ОС за междупроцесна синхронизация;
  - разкрие как описаните примитиви се категоризират според целевата функция;
  - илюстрира практически примери в контекста на Posix Pthread API и Линукс ОС

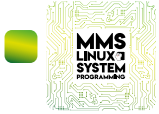


Синхронизация на процеси с фокус върху подходи за блокиране и взаимно изключване

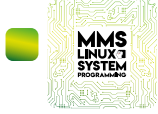


Предпоставки за възникване на междупроцесни състезания

- Примери:
  - **съревнование** за боравене с краен набор физически или логически средства (например: входно-изходни устройства, заемане на изчислителни блокове и др.);
  - **комуникация посредством споделени сегменти от паметта**, при които съществува риск от компрометиране на целостта на данните при множествен достъп;
  - отработване на поредица действия в резултат на възникване на конкретно системно събитие и неговото сигнализиране;
  - **асинхронно постъпване на нови задания** и нужда от планиране на работния товар (предизвикателства, съотнесени към дисциплините за системно планиране);
  - **паралелно изпълнение** на **конкуриращи** или **кооперирващи** се процеси или потоци

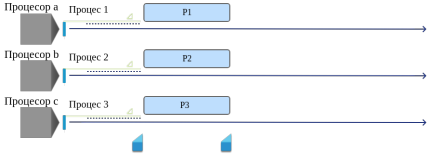


Да си припомним възможни сценарии за развитието на множество задачи съобразно политиката за планиране



Планиране на постъпващите процеси: примери

Времева линия на изпълнение и активиране на диспечера на процеси при едноядрена система

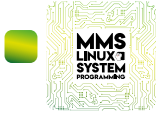


Времева линия на изпълнение и активиране на диспечера на процеси при многоядрена система

pic. based on `https://en.wikipedia.org/wiki/Scheduling_(computing)`

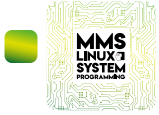
Въведение в междупроцесната синхронизация

- Дефиниция и потребности:
  - синхронизацията между заданията е необходима за имплементиране на взаимодействие между група процеси, като **процесът на координация** бива **пряк** и **косвен**;
  - основание за съгласуване при сътрудничещи си програми е необходимостта от структуриране на подзадачите в процеса на работа и своевременния обмен на данни, без които синхронът би бил невъзможен;
  - предпоставка за координация при съревновавщи се задачи е изискването да се запази интегритетът на споделените данни и да се гарантира ексклузивен достъп до обектите, защитени с критични секции;



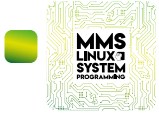
Методика за координация на процеси

- **Съгласуването бива:**
  1. **пряко**, когато се ползват примитиви и механизми за предотвратяване на нежелано блокиране и указване на правилата за взаимодействие и взаимно изключване;
  2. **непряко**, когато резултатът от обработката на едно задание е необходимо условие за протичане на последващо без явна взаимовръзка или състезание за достъп до общ ресурс;
- **Примери:**
  - по т.1 - набор от **процеси производители** и един или повече **процеси потребители**, свързани с общ краен буфер елементи;
  - по т.2 - **разпаралеляване на алгоритъм** на отделни взаимосвързани блокове или **конвейерна обработка**



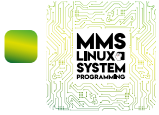
Междупроцесна синхронизация: подходи

- Разновидности:
  - **софтуерни (програмни решения)** - ползвайки логически конструкции и алгоритми за осигуряване на синхрон посредством **програмно изчакване** като вече споменатия алгоритъм на Peterson;
  - **хардуерни подходи**, при които взаимното изключване или планиране на изпълнението се решава чрез **апаратни блокове**, структури или устройства, подсигуряващи **критична секция**



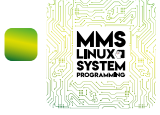
Междупроцесна синхронизация: необходими условия

- Прилаганите подходи трябва да гарантират:
  - **коректно и надеждно изграждане на критична секция** и управление на достъпа, ако това е наложително за манипулиране на ограничен логически или физически ресурс;
  - възможност за навлизане в критичната секция в **обозримо бъдеще време** без риск от безкрайно отлагане на очакващо задание;
  - **безпристрастен механизъм** за селектиране на чакащ достъп процес и контролиране на времето, за което той борави с критичния споделен ресурс;
  - **справедлива политика за планиране на процесите**, така че да се предвиди възможност на отложено или изчакващо задание да се изпълни;
  - не се допускат други ограничения или предпоставки, които могат да повлияят върху механизмите за взаимно изключване и/или диспечериране



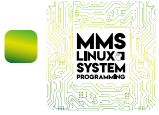
Роля на операционната система и системното програмно осигуряване

- Отговорности:
  - подсигуряване на **набор от подходящи примитиви** за синхронизация и комуникация, съобразени със спецификите на микроархитектурата;
  - коректна реализация на **подход за взаимно изключване** и справедлива селекция между състезаващите се процеси или потоци;
  - осигуряване на **приемлива ефикасност** при боравенето със способите за синхронизация: както от гледна точка на имлементацията на процедурите по заключване и блокиране, така и по отношение на влиянието върху алгоритмите на ОС за планиране при многоядрени или многопроцесорни системи

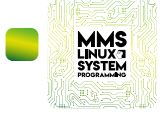


Програмни техники за предпазване от състезания

- На високо ниво се открояват:
  - **реентрантност** - блокът операции може да бъде частично изпълнен и прекъснат, рекурсивно извикан повторно или изпълнен паралелно от друг поток без това да повлияе негативно върху коректността на крайния резултат;
  - локализиране на променливите - всяка нишка разполага със свое локално работно копие **thread-local storage**;
  - неизменими обекти **immutable objects** - не подлежат на промяна след създаване (конструкция);
  - взаимно изключване **mutual exclusion** при множество процеси или потоци, при които се изгражда критична секция, взаимно изчакване и явно арбитриране на четене и запис в паметта;
  - атомарни операции над променливи **atomic operations** - изпълняват се без прекъсване и без вмешателство от страна на друг поток, като при многоядрени системи терминът предполага заключване на достъпа до завършване на операцията

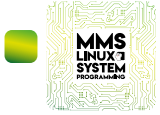


Какво значение се влага в определенията реентрантна (reentrant), синхронизирана многонишкова (thread-safe) и идемпотентна (idempotent) процедура или последователност?



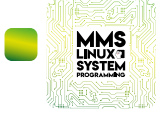
Реентрантен, защитен многопоточен и инвариантен

- Определение и смислово разграничение:
  - **реентрантен** - запазва се контекстът и интегритета на изчисленията, ако процедурата бъде прекъсната и извикана повторно в различен контекст или от друг блок за обработка;
  - **синхронизиран многонишков** - реализира критична секция и блокира опит за конкурентен достъп от страна на други потоци, ако един вече е навлязъл в защитения блок;
  - **идемпотентен** - инвариантен по повод на входните аргументи в етапа на изпълнение.

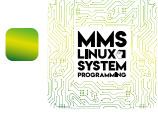


Примери

- Приема се еквивалентно тълкуване на функция и процедура:
  1. **реентрантна функция** - `int func(int x) {return 2 * x;}`, като още примери може да откриете тук: `https://man7.org/linux/man-pages/man7/signal-safety.7.html`
  2. **thread-safe** - функции, при които се прилагат инструменти за блокировка, координация и взаимно изчакване, включително и в аспект работа с паметта;
  3. **идемпотентна функция** - релация на тъждественост: f(x) = x.

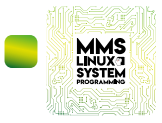


А как изглежда една нереентрантна процедура? Всичко това изглежда обръкващо...



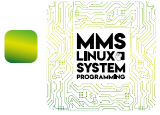
Нереентрантна процедура: илюстрация

```c
/* non-reentrant and non-thread-safe */
void global_sum(int* x, int y) {
    int res;

    /* x points to a global variable */
    /* no multithreaded locking present */
    res = *x;
    res += y;
    *x = res;
}

/* reentrant sum */
int reentrant_sum(int a, int b) {
    int res = 0;

    /* all args live on the stack */
    res = a + b;
    return res;
}
```



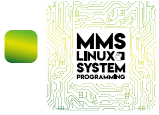
Реентрантен не означава thread-safe или атомарен. Защо?

- Защото се влага различен смисъл и обхват на приложение:
  - **реентрантността визира разполагане на входните и изходните аргументи в стека**, като така се локализира регионът на взаимодействие, елиминират се обръщения към глобални променливи;
  - **реентрантността не е задължително синхронизирана**, защото обикновено се обособява блок от обработки, които сами по себе си не е задължително да включват заключване, взаимно изчакване или многопоточна блокировка;
  - **реентрантността не е гарантирано атомарна**, защото повечето реални операции са изградени от изпълнението на поредица от повече от една процесорни инструкции, която няма отношение към механиката на апаратните прекъсвания и не налага изисквания за това в повечето случаи



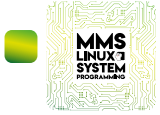
Необходими условия за реентрантност

- Изисквания:
  - да не се правят референции към глобални или статични променливи;
  - да не се видоизменя автономно изпълнимият код в хора на работа;
  - тялото на процедурния блок да не съдържа обръщения към други нереентрантни функции

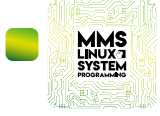


Реентрантни функции: приложение

- Ползват се при:
  - разработка на модули в състава на операционната система: платформени и устройствени драйвери, като добра практика при програмиране на **обработка на прекъсвания (interrupt handlers)** или предотвратяване на нежелани последствия при наличие на прекъсване и приоритетното му обслужване от друг подпроцес;
  - в системното програмиране - за минимизиране на нежелани спонтанни изменения на стойността или съдържанието на общореферирани променливи - пример с обработка на възникнал сигнал **(signal handler)**;
  - **в контекста на ОС понятието реентрантност се надгражда и разграничава от понятието атомарност** и податливост на прекъсване от диспечера, което добавя ново ниво на осмисляне и техника на програмиране

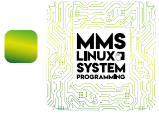


Реалното многообразие от процеси и развитието им ограничава използването на пасивни техники. Създаването на реентратни процедури невинаги е възможно и оправдано на високо ниво. Необходимо е да се запознаем с примерни методи и средства за междупроцесна координация.

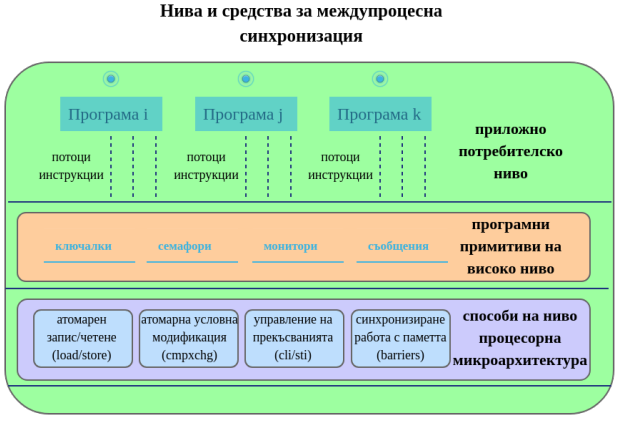


Разновидности примитиви за синхронизация на процеси

- Категоризация:
  - **апаратни и/или софтуерни прекъсвания** с пряко влияние върху избора на активно задание и времето му за изпълнение от процесора;
  - **ключалки** - обекти, опериращи най-често върху файлови дескриптори и ползвани за координиране достъпа до споделени файлови обекти, като се разделят на общи (за четене) и ексклузивни (когато притежаващият процес извършва и модификация върху данните);
  - **семафори** - обекти, ползвани за подсигуряване на взаимно изключване на два или повече процеса, като се делят на двоични и броячни;
  - **мутекси и условни променливи** - обезпечават блокиране на потоци по отношение на изпълнявана обработка или боравене със системни ресурси до удовлетворяване на определено условие и съпътстващото сигнализиране



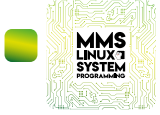
Нива на синхронизация: диаграма



pic. based on `https://en.wikipedia.org/wiki/Synchronization_(computer_science)`

Взаимно изключване с хардуерни способи

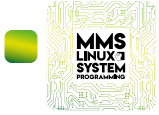
- Открояват се решения с:
  - употреба, обработка и контрол върху прекъсванията - тук блокирането би могло да се осъществи чрез временна забрана за обслужване на прекъсвания и превключване на контекста (**cli** и **sti** инструкции при Intel x86);
  - машинни команди за атомарно изпълнение (от вида на **TS - atomic test and set**), при които критичната секция се реализира чрез атомарно (непрекъсваемо) установяване на променлива или аналогична операция чрез регистрови структури на процесора, (например: **cmpxchg: compare and exchange** or **TS: test and swap**)



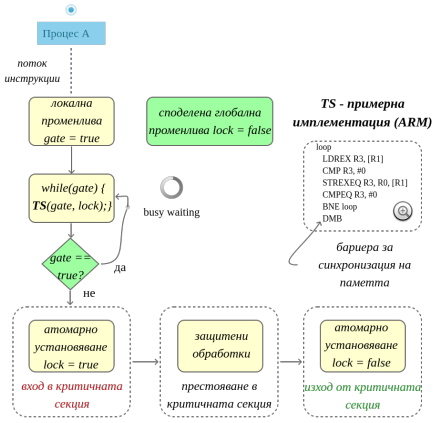
Координация с манипулиране на прекъсванията

Набор от архитектурно специфични функции на най-ниско ниво:

```
/* Pseudocode examples */
/* Prevents preemption */
void lock() {
    /* A special function, which calls
    processor-specific instructions */
    interrupts_disable();
}

/* Reenables preemption */
void unlock() {
    interrupts_enable();
}
```



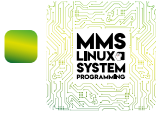
Атомарно взаимно изключване: диаграма



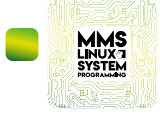
pic. based on `https://en.wikipedia.org/wiki/Test-and-set`

Синхронизация чрез апаратни прекъсвания: недостатъци

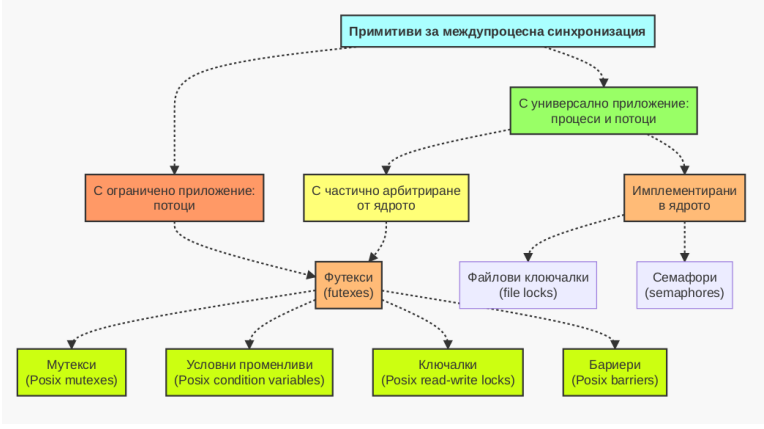
- Най-съществените са:
  - **необходимост от превключване на контекста** и извършване на привилегирована операция от името на операционната система, което внася допълнителна латентност;
  - **риск от монополизиране на процесора**, когато злоупотребяваща програма реши да забрани за дълъг период детекцията и обслужването на прекъсвания - резултатът е неработоспособна система;
  - **обръкваща приложимост на подхода върху многоядрени или многопроцесорни системи**, при които последователността, определяща критичната секция, би могла да се изпълни едновременно върху различни ядра;



Добре, а какви програмни примитиви на високо ниво се използват?

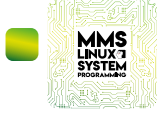


Класификация на примитиви за синхронизация



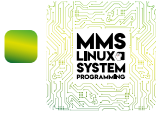
pic. based on `https://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf`

Част I - Синхронизация на процеси посредством условно заключване със средства от високо ниво



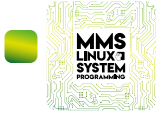
Условно активно заключване: наивен подход

- Техника с централизиран флаг и активно чакане (busy waiting):
  - примерно решение е показано във вече описания **алгоритъм на Peterson**;
  - разчита се на **споделен Булев флаг** и **специализирани инструкции за атомарно** прочитане, условно записване и проверка на стойността на флага (пример: **ldrex** и **strex** инструкции при ARM);
  - критичната секция се изгражда на база на устнановяване на стойността на общия флаг;
  - при отчитане на вече инициализирана променлива - **конкуриращ се поток се блокира**, изпълнявайки периодична проверка на управляващата стойност и **изчакване в празен цикъл**



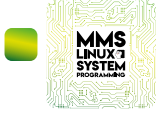
Условно активно заключване: подобрение

- Би могло да включва:
  - вместо безсмислено заемане на процесора при изчакване се приема **техника на обръщение към диспечера** чрез **системно извикване (yield)**;
  - **yield** поставя извикващия поток в опашка от чакащи процеси;
  - **Линукс предприема хибриден подход**, съчетаващ **първоначален цикъл на busy waiting** и **последващо отнемане на процесора** или блокиране посредством **futex (fast userspace mutex)**;
  - множество **синхронизационни конструкции от високо ниво** са имплементирани с помощта на futex операции като **mutex**, **condition variables** и **barriers**;

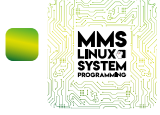


Активно заключване: обобщение

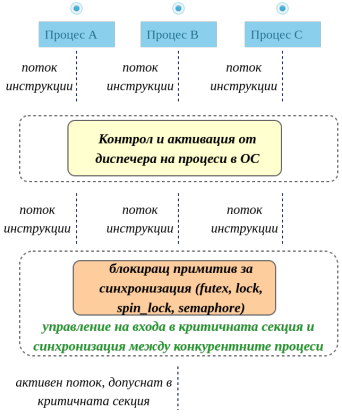
- Особености:
  - изисква **референция към споделена променлива или регион памет** (при множество процеси);
  - налага **ползване на специализирани процесорни инструкции**, включително **барирери на паметта (memory barriers)**;
  - поради евентуално **превключване на контекста** би могло да се окаже неефикасно при множество съревноваващи се процеси или потоци;
  - необходимо е не само **атомарно четене и запис**, но и манипулиране на опашките на системния диспечер;
  - подходът със забраняване на обслужването на прекъсванията се предпочита при компактни критични секции с кратко планирано време за изпълнение и малко на брой конкурентни процеси;
  - съвременните ОС като Линукс предоставят **хибриден подход** на условно заключване, подобен на описания, за да се подобри системната производителност



Условно заключване и блокиране: програмни примитиви - футекси, ключалки и семафори



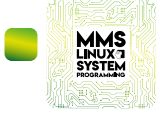
Управление на достъпа до критична секция: диаграма



pic. based on `https://en.wikipedia.org/wiki/Synchronization_(computer_science)`

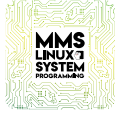
Взаимно изключване чрез futex

- Дефиниция и постановка:
  - фундаментална логическа абстракция за имплементиране на различни видове синхронизационни примитиви;
  - позволява голяма част от операциите по взаимно арбитриране да се развият в **потребителски режим** с частичен помощен арбитраж от ядрото;
  - ползва споделен регион виртуална памет между процесите и всяка инстанция се представя обикновено чрез 32-битово цяло число;
  - всеки процес изобразява споделения футекс обект с уникален виртуален адрес в своето адресно пространство, но футексът реферира една и съща физическа локация;
  - **futex системно извикване** и обръщение към ядрото се прави само когато процес ще се блокира за по-дълъг период от време, тогава ядрото на ОС е отговорно за по-нататъшната нотификация

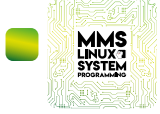


Futex: механизъм на реализация

- Механика и приложение:
  - състоянието на обекта ключалка се изобразява с атомарно установяван и проверяван флаг или числена стойност;
  - всеки процес или поток би могъл да измени съдържанието на споделения флаг, ползвайки специализирани неделими инструкции от рода на посочените test and set варианти;
  - тези инструкции се развиват в потребителски режим и ядрото на ОС не поддържа метаданни за описание на състоянието на ключалката;
  - когато втори процес желае да модифицира вече изменен футекс обект, той предава прочетената стойност на обекта и желаната такава с помощта на системно извикване към ядрото, след което бива блокиран;
  - само в описаната ситуация на блокиране ядрото поддържа вътрешни записи, отразяващи съдържанието на футекс обекта, за да управлява привеждането на процес в блокирано състояние или в състояние на готовност;
  - процес, освобождаващ обекта ключалка, трябва да възвърне оригиналната целочислена стойност, отразена от футекса - тогава ядрото взема предвид това и събужда чакащ процес

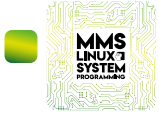


Повече информация за futex бихте могли да откриете тук:
`https://man7.org/linux/man-pages/man2/futex.2.html`



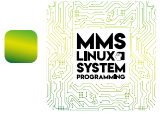
Posix ключалки

- Дефиниция:
  - логически обект, подсигуряващ взаимно изключване на конкуриращи се процеси и организиране на критична секция;
  - само един процес или поток може да заяви свободна ключалка, като след като това стане, останалите желаещи достъп, ще преминат в **режим на активно очакване (spinning)**;
  - недостатък е, че блокираните потоци изпълняват активно чакане (busy looping), което лимитира обхвата на приложение;
  - препоръчва се заместването на spinlock обекти с мутекси що се касае до изграждане на критична секция;

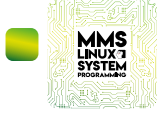


Posix Pthread ключалки: приложна секция

- Полезни API функции:
  - `int` pthread_spin_init (pthread_spinlock_t *lock, `int` pshared) - инициализация на обект ключалка;
  - `int` pthread_spin_lock (pthread_spinlock_t *lock) - блокиращо изчакване на придобиване на ключалка за навлизане в критична секция;
  - `int` pthread_spin_trylock (pthread_spinlock_t *lock) - неблокиращ аналог на горната функция;
  - `int` pthread_spin_unlock (pthread_spinlock_t *lock) - освобождаване на ключалка и позволяване на диспечера да пропусне друг очакващ процес;

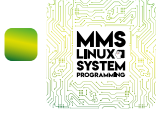


Posix Pthread spin_locks: разглеждане на пример:
`https://man7.org/tlpi/code/online/dist/threads/thread_incr_spinlock.c.html`



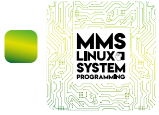
Критична секция чрез семафори

- Дефиниция:
  - въведени от Edsger Wybe Dijkstra за пръв път през 60-те години на 20-ти век;
  - представляват променливи с две позволени операции: **P** и **V**;
  - различават се **двоични семафори** (приемащи стойност 0 и 1) и **броячни такива** (приемащи неотрицателни целочислени стойности);
  - броячните семафори биха могли да служат за контролирано управление на достъпа на множество процеси до краен набор системни ресурси



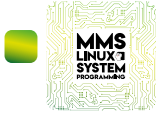
Семафорни операции

- Пояснение:
  - **P()** - атомарна операция, при която се изчаква семафорът да приеме положителна ненулева стойност и след това го декрементира с 1;
  - **V()** - неделима операция, при която стойността на семафора се инкрементира с 1, след което се разрешава блокирана P() функция

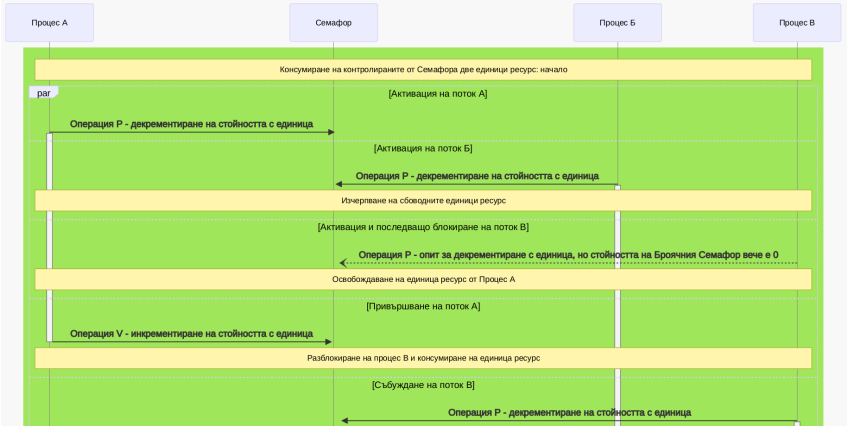


Семафори: механизъм на имплементация

- **Реализацията на P() се базира върху:**
  - атомарни инструкции за модифициране стойността на семафорната променлива;
  - ако стойността е 0, то се прави запис на извикващия процес в привързана опашка и процесът се блокира;
  - ако стойността не е 0, тя се декрементира и извикващият процес бива допуснат в защитената секция;
- **Реализацията на V() се гради върху:**
  - атомарни инструкции за изменение стойността на семафорната променлива;
  - ако привързаната опашка от блокирани процеси не е празна, се изважда процес и се привежда в състояние на готовност;
  - инкрементира се стойността на променливата



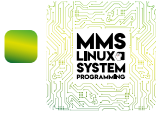
Диаграма на взаимодействието чрез семафори



pic. based on `https://en.wikipedia.org/wiki/Semaphore_(programming)`

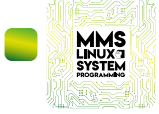
Posix Pthread семафори: приложна секция

- Полезни API функции:
  - `int` sem_init(sem_t *sem, `int` pshared, `unsigned int` value) - инициализация на семафорен обект;
  - sem_t *sem_open(`const char` *name, `int` oflag) - заявка за достъп до семафорен обект;
  - `int` sem_close(sem_t *sem) - освобождаване референция към активен семафор;
  - `int` sem_post(sem_t *sem) - отключване на семафор и сигнал за разблокиране на чакащ процес;
  - `int` sem_wait(sem_t *sem) - блокиращо изчакване семафор да бъде отключен;
  - `int` sem_trywait(sem_t *sem) - неблокиращ аналог на горната функция;
  - `int` sem_timedwait(sem_t * restrict sem, `const struct` timespec * restrict abs_timeout) - блокиращо изчакване с времеви лимит;



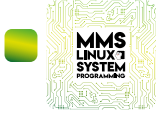
Posix Pthread semaphores: разглеждане на пример:

`https://man7.org/tlpi/code/online/dist/psem/psem_create.c.html`



Условна критична секция: потенциални проблеми

- Съществува риск от:
  - **възникване на мъртва хватка** с безкрайно отлагане на процеси или потоци;
  - **инверсия на приоритетите** - когато високоприоритетна задача бива отложена поради факта, че нископриоритетна е преминала в състояние на изпълнение и е навлязла в критична секция. По този начин се е отложила активацията на високоприоритетната задача, която се блокира пред същата критична секция. Така високоприоритетната не би могла да завърши в рамките на очаквания времеви интервал, в резултат на което тя пропада и може да доведе до срив на системата.

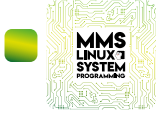


Илюстрация на взаимно блокиране

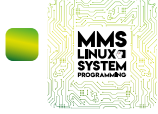
pic. based on `https://en.wikipedia.org/wiki/Deadlock`

Предпоставки за възникване на мъртва хватка

- Посочват се:
  - **некоректно взаимно изключване на процеси** - споделеният ресурс може да се ползва ексклузивно само от един потребител в даден момент;
  - **блокиране в рамките на критичната секция** - процес, навлязъл в критична секция и блокиращ други, навлиза в състояние на изчакване освобождаването на друг системен ресурс;
  - **липса на надзор при разпределението на средствата** - процес, зает обект, може да го освободи само доброволно;
  - **кръгово блокиращо очакване** - верига от процеси, при която всеки очаква освобождаването на обект, зает от предходния

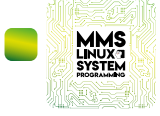


Съществува ли удачен алгоритмичен подход за детекция на мъртва хватка?



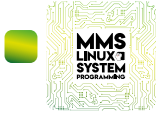
Граф на разпределение на ресурсите

- Моделиране на системата и постановка:
  - крайно множество от ресурси, представено чрез вектор: $< R_i >$;
  - краен набор от процеси ползватели, представено чрез вектор: $< P_i >$;
  - съществуват: N_i инстанции от всеки ресурс: R_i;
  - всеки поток инструкции употребява ресурса чрез следната последователност: **заявка** $->$ **ползване** $->$ **освобождаване**

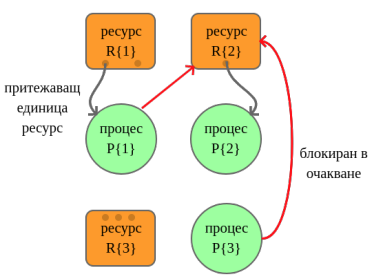


Граф на разпределение на ресурсите: продължение

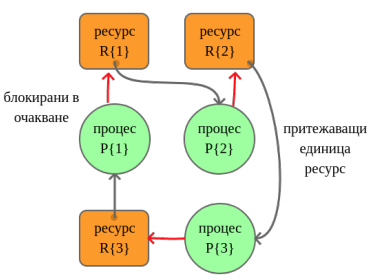
- **Построява се граф посредством:**
  - двете посочени множества на ресурсите R_i и процесите P_i;
- **Откроват се два вида насочени ребра:**
  - на **заявяване**: $T_i -> R_j$;
  - на **присвояване**: $R_j -> T_i$



Илюстрация на примерни граф структури



Сценарий А - граф на разпределение на ресурсите без цикли (без мъртва хватка)

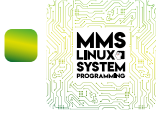


Сценарий Б - граф на разпределение на ресурсите с кръгов цикъл (мъртва хватка)

pic. based on `https://inst.eecs.berkeley.edu/~cs162/fa19/static/lectures/11.pdf`

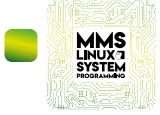
Алгоритъм за откриване на мъртва хватка

- Базира се върху:
  - структури от данни, чрез които се строи **динамичен граф на заетостта на ресурсите**
- Подсигуряване на следните зависимости:
  - две възможни състояние на ресурсите: **разпределени** или **налични**;
  - никой процес не може да заяви повече от наличните ресурси или повече от тотално заявените

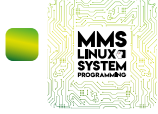


”Алгоритъм на Банкера”

- **Разработен от Edsger Dijkstra и разчитащ на:**
  - **вектор на процесите**: $< P_n >$;
  - **вектор на наличните ресурси**: $< R_m >$;
  - **матрица на допустимите максимални заявки за всеки процес**: Q_{nm};
  - **матрица на разпределените ресурси за всеки процес**: P_{nm}
- **Повече информация:**
  - `https://en.wikipedia.org/wiki/Banker%27s_algorithm`

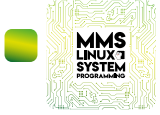


Deadlock на процеси все пак се случва в реалността. Има ли и други подходи?

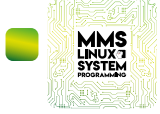


Стратегии за предотвратяване на мъртва хватка

- Направления:
  - **програмни подходи** за предотвратяване при фазите на проектиране и разработка;
  - **стратегия на избягване** с повлияване върху разпределението на ресурсите (опционално без необходимост от блокиране);
  - въвеждане на **системни механизми за детекция** и възстановяване на работоспособно състояние **(watchdog daemon)**

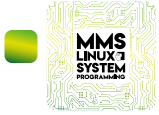


Част II - Синхронизация на потоци със средства от високо ниво

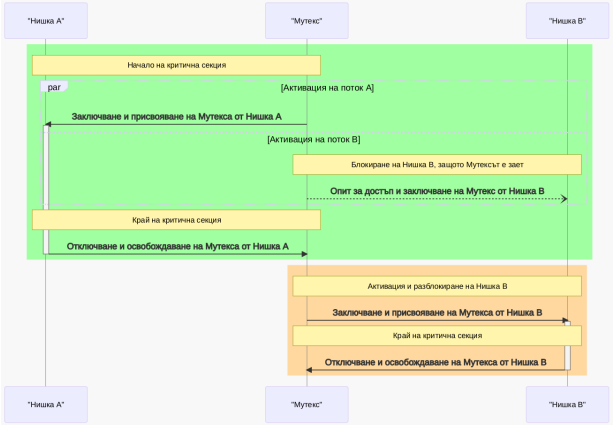


Posix Pthread мутекси

- Определение:
  - аналог на семафорни примитиви, ползвани за координация на конкуриращи се потоци;
  - блокирането и изчакването се осъществяват на ниво потоци от инструкции в контекста на един процес



Визуално представяне на взаимодействие чрез мутекси



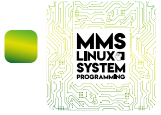
pic. based on `https://en.wikipedia.org/wiki/Lock_(computer_science)`

Смислово разграничаване между семафор и мутекс

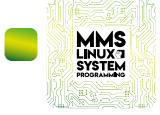
- **Характеристики на семафорите:**
  - **споделени обекти**: множество процеси или потоци могат да ползват референция към един семафор и да изменят стойността;
  - **с по-широка област на приложение**: позволяват както имплементиране на критична секция и взаимно блокиране, така и ефикасно управление на разпределение на достъпа до ограничен набор ресурси единици;
  - **генерализиран примитив с възможност за реализация и на форма на междупроцесна комуникация**: броячните семафори намират приложение и като примитив за сигнализиране за настъпило събитие;
- **Характеристики на мутексите:**
  - **ексклузивно достъпвани обекти**: с асоциирана принадлежност и строго определена последователност на употреба;
  - **с конкретна ограничена област на приложение**: служат за взаимно блокиране на конкуриращи се потоци;
  - **с по-неефикасно бързодействие при употреба**: входът и изходът от критичната секция (блокирането и активацията на потоците) са изцяло под контрола на системния диспечер на ОС

Posix Pthread мутекси: приложна секция

- Полезни API функции:
  - pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;;
  - `int` pthread_mutex_init (pthread_mutex_t * restrict mutex, `const` pthread_mutexattr_t * restrict attr) - инициализация на мутекс;
  - `int` pthread_mutex_destroy(pthread_mutex_t *mutex) - освобождаване на заделен мутекс;
  - `int` pthread_mutex_lock(pthread_mutex_t *mutex) - блокиращо изчакване мутекс да бъде освободен;
  - `int` pthread_mutex_trylock (pthread_mutex_t *mutex) - неблокиращ аналог на горната функция;
  - `int` pthread_mutex_unlock(pthread_mutex_t *mutex) - отключване на мутекс обект и сигнализиране за разблокиране на очакващ поток

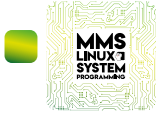


Posix Pthread мутекси: разглеждане на пример:
`https://man7.org/tlpi/code/online/dist/threads/thread_multijoin.c.html`

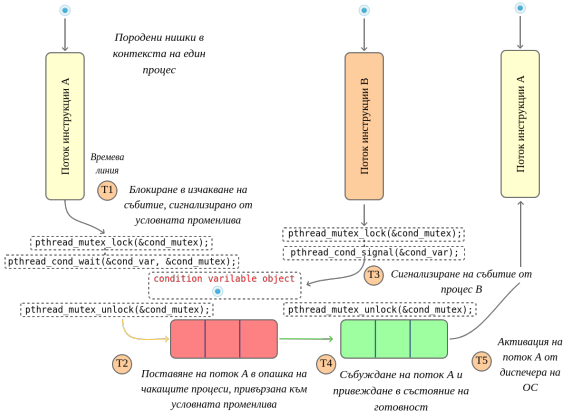


Условни променливи

- Дефиниция и предназначение:
  - подсигуряват синхронизиране на потоци, но на база **стойност на споделена конструкция**, а не чрез манипулиране на достъпа до нея;
  - служат не само за взаимно блокиране, но и за **форма на междупоточна комуникация**;
  - нотификацията за обновяване на целевата стойност, блокирането и разблокирането на конкуриращите се нишки е поверено на ОС;
  - условните променливи са **имплементирани посредством футекси** и **елиминират необходимостта от polling** (активно периодично проверяване на стойност на общ обект данни)



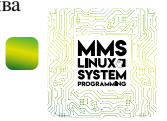
Онагледяване на взаимодействие с условни променливи



pic. based on `https://en.wikipedia.org/wiki/Observer_pattern`

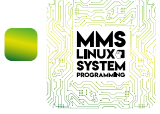
Posix Pthread условни променливи: приложна секция

- Полезни API функции:
  - pthread_cond_t cond = PTHREAD_COND_INITIALIZER
  - `int` pthread_cond_init (pthread_cond_t * restrict cond, `const` pthread_condattr_t * restrict attr) - инициализиране на условна променлива;
  - `int` pthread_cond_signal (pthread_cond_t *cond) - сигнализиране и разблокиране на само един от изчакващи потоци, когато условната променлива е достигнала желана стойност;
  - `int` pthread_cond_broadcast (pthread_cond_t *cond) - сигнализиране и разблокиране на всички изчакващи потоци, когато условната променлива е достигнала желана стойност;
  - `int` pthread_cond_wait(pthread_cond_t * restrict cond, pthread_mutex_t * restrict mutex) - изчакване и блокиране на поток докато условната променлива не приеме зададена стойност;
  - `int` pthread_cond_destroy (pthread_cond_t *cond) - освобождаване на обект условна променлива



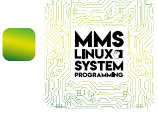
Posix Pthread условни променливи: пояснение

- Препоръчителна програмна последователност:
  - всеки конкурентен поток изпълнява първоначално **mutex_lock**, последван от **cond_wait**;
  - ако условната проверка върне лъжа, тогава потокът се блокира пред условната променлива и се излиза от критичната секция с **mutex_unlock**;
  - при промяна на условната променлива от друга нишка и извършена сигнализация, чакащ поток се избира и събужда, като взаимоизключващата блокировка се подновява



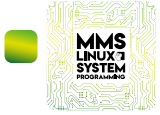
Posix Pthread условни променливи: разглеждане на пример:

`https://man7.org/tlpi/code/online/dist/threads/prod_condvar.c.html`



Posix Pthread ключалки за четене и запис

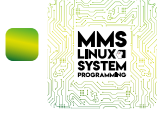
- Дефиниция:
  - различават се от класическите обекти активни ключалки и служат за логическо разделяне на потоците на **”читатели”** и **”писатели”**;
  - допускат едновременното развитие на множество нишки, достъпващи споделени обекти в паметта само за четене или само на един процес, които желае да нанесе промяна на споделеното съдържание;
  - аналогични са на броячни семафори по отношение на читателите;
  - аналогични са на двоични семафори що се отнася до потоците писатели;



Posix Pthread ключалки за четене и запис: приложна секция

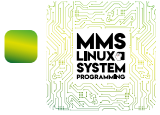
- Полезни API функции:
  - pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER
  - `int` pthread_rwlock_init (pthread_rwlock_t * restrict rwlock, `const` pthread_rwlockattr_t * restrict attr) - инициализация на обект ключалка;
  - `int` pthread_rwlock_destroy (pthread_rwlock_t *rwlock) - освобождаване на ключалката;
  - `int` pthread_rwlock_rdlock (pthread_rwlock_t *rwlock) - блокиращ опит за активиране достъп само за четене посредством ключалката;
  - `int` pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock) - аналог на горната функция, но без блокиране при неуспех за ”отключване” на ключалката;
  - `int` pthread_rwlock_wrlock(pthread_rwlock_t *rwlock) - блокиращ опит за получаване достъп за запис чрез ключалката;
  - `int` pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock) - аналог на горната функция, но без блокиране при неуспех за ”отключване” на ключалката;
  - `int` pthread_rwlock_unlock(pthread_rwlock_t *rwlock) - функция за ”отключване” на ангажиран обект ключалка

Posix Pthread ключалки за четене и запис: разглеждане на пример:
`https://man7.org/tlpi/code/online/dist/threads/thread_incr_rwlock.c.html`

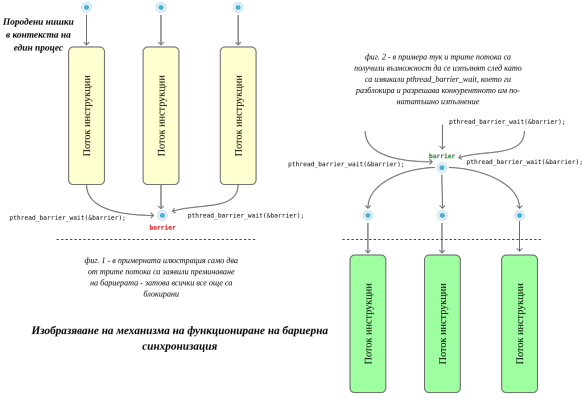


Posix Pthread бариери

- Дефиниция и предназначение:
  - обекти, които подсигуряват взаимно блокиране на конкретен набор потоци докато всеки от един от тях не заяви желание да продължи изпълнението след региона, маркиран от бариерата;
  - след като така описаното граничното условие е удовлетворено и всички нишки са достигнали точката на бариерата - диспечерът на ОС разрешава по-нататъшното паралелно изпълнение на тези потоци, като редът на активация е недетерминиран



Диаграма на механизъм с бариерна синхронизация

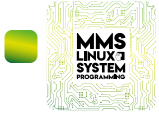


Изобразяване на механизма на функциониране на бариерна синхронизация

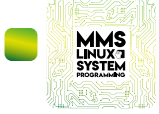
pic. based on `https://en.wikipedia.org/wiki/Barrier_(computer_science)`

Posix Pthread бариери: приложна секция

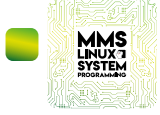
- Полезни API функции:
  - `int pthread_barrier_init ( pthread_barrier_t * restrict barrier , const pthread_barrierattr_t * restrict attr , unsigned count)` - инициализация на обект бариера;
  - `int pthread_barrier_destroy ( pthread_barrier_t * barrier )` - освобождаване на референция към създадена барирера;
  - `int pthread_barrier_wait ( pthread_barrier_t * barrier )` - изчакване и сигнализация на заявка за навлизане в критична секция, маркирана от бариерата



Posix Pthread бариери: разглеждане на пример:
`https://man7.org/tlpi/code/online/dist/threads/pthread_barrier_demo.c.html`

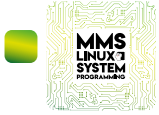


В следващата лекция предстои да обсъдим и механизми за междупроцесна комуникация



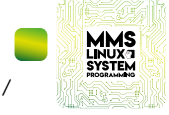
Бележки по материалите и изложението

- 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/wiki/Reentrancy_(computing)
- https://en.wikipedia.org/wiki/Synchronization_(computer_science)
- https://en.wikipedia.org/wiki/Thread_safety
- https://lwn.net/Articles/823513/
- https://opensource.com/article/19/4/interprocess-communication-linux-storage
- https://tldp.org/pub/Linux/docs/ldp-archived/linuxfocus/English/Archives/lf-2003_01-0281.pdf
- https://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf
- https://man7.org/linux/man-pages/man7/futex.7.html
- https://man7.org/training/download/lusp_pshm_slides.pdf
- https://inst.eecs.berkeley.edu/~cs162/fa19/static/lectures/11.pdf
- https://inst.eecs.berkeley.edu/~cs162/fa19/static/lectures/4.pdf
- https://www.gnu.org/software/libc/manual/html_node/Nonreentrancy.html
- https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf
- https://developer.arm.com/documentation/dht0008/a/arm-synchronization-primitives/exclusive-accesses/ldrex-and-strex



Благодаря Ви за вниманието!