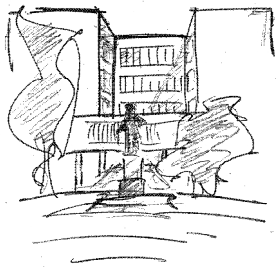
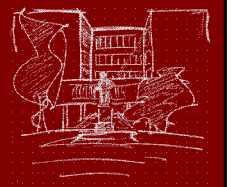
Развој софтвера 10



Саша Малков
Универзитет у Београду
Математички факултет
2023/2024

[P290]
Развој софтвера
Саша Малков



Тема 14

Конкурентно програмирање

Основни појмови

- Конкурентно извршавање
- Паралелно извршавање
- Дистрибуирано извршавање
- Процес
- Промена контекста
- Нит

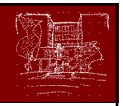
Конкурентно извршавање

- Два посла T1 и T2 се извршавају ***конкурентно*** ако се извршавају у ***истим временским оквирима***, али ***није унапред позната њихова међусобна временска лоцираност***
- "...у истим временским оквирима..."
  - нпр. два процеса се покрећу асинхроно један за другим...
- "...није унапред позната њихова међусобна временска лоцираност..."
  - знамо да ће се они извршавати у истом временском интервалу, али не знамо којим редом ће добијати процесорско време...

Конкурентно извршавање (2)

- Два посла T1 и T2 се извршавају *конкурентно* ако се извршавају у истим временским оквирима, али није унапред позната њихова међусобна временска лоцираност:
  - T1 може да почне и заврши се пре почетка T2
  - T1 може да почне пре и заврши се после започињања T2
  - T2 може да почне пре започињања посла T2 и заврши се после завршавања посла T2
  - T2 може да почне после започињања посла T2 и заврши пре завршавања посла T2
  - ...

Конкурентно извршавање (3)



- На српском језику, због игре речи и значења, може да се разуме да конкурентни токови извршавања представљају ***конкуренцију*** једни другима
  - то је у овом контексту углавном исправно
  - конкурентни послови се међусобно боре за ресурсе
- Дословни превод би пре био
  - ***истовремени***, који постоје у исто време

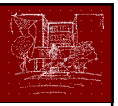
Конкурентно извршавање (4)

- Конкурентно извршавање је логички концепт
  - о пословима се размишља као да се извршавају истовремено
  - води се рачуна о усклађивању и синхронизацији

Паралелно извршавање

- Два посла се извршавају ***паралелно*** ако постоји период времена у коме су (дословно) истовремено активни
  - паралелно извршавање је могуће само ако имамо на располагању више извршних јединица (процесора или језгара)

Паралелно извршавање (2)



- Паралелно извршавање је имплементациони концепт
  - подразумева се да послови раде исттовремено и да користе различите или дељиве ресурсе

Конкурентно и / или паралелно

- Конкурентност подразумева неодређеност временског односа
- Паралелност подразумева предодређеност временског односа
- Конкурентни програми у крајњем исходу могу да раде секвенцијално, или део по део секвенцијално, а паралелни не могу, већ *морају* да раде истовремено
- Конкурентни послови могу, али не морају да се извршавају паралелно
- Паралелни послови могу али не морају да буду конкурентни
  - паралелни послови нису конкурентни ако раде на одвојеним (под)скуповима података и није потребна никаква синхронизација

Конкурентно и / или паралелно (2)

- *The Art of Concurrency, Clay Breashears:*
  - Програми су ***конкурентни*** ако су истовремено ***у стању извршавања***
  - Програми су ***паралелни*** ако се истовремено ***извршавају***

Конкурентно и / или паралелно (3)

- *Конкурентно* извршавање је могуће и на једној извршној јединици (процесору)
  - програми деле процесорско време
  - тим дељењем управља ОС
- Паралелно извршавање захтева више извршних јединица (процесора)

Конкурентно програмирање / Појмови

Дистрибуирано извршавање

- Два посла се извршавају ***дистрибуирано*** ако раде у различитим адресним просторима
  - ово је општија дефиниција, која тежи да изједначи (апстрахује) дистрибуираност између више рачунара и дистрибуираност између компоненти (процесора) истог рачунарског система
  - старије дефиниције подразумевају различите рачунарске системе, али то није сасвим у реду у случају када више рачунарских система дели радну меморију – тада се послови извршавају слично као на једном рачунару са више процесора, тј. конкурентно

Универзитет у Београду - Математички факултет

Конкурентно програмирање / Мотивација

Паралелизација

- Паралелизацијом називамо писање (или прилагођавање) програма тако да користе могућности паралелног извршавања
- Уобичајено је да се исти термин употребљава и за писање конкурентних програма
- У оквирима овог курса, паралелизацијом ћемо називати писање програма уз примену конкурентног извршавања више процеса или нити

Универзитет у Београду - Математички факултет

Конкурентно програмирање / Мотивација

Мотивација

- Паралелизација, тј. подела посла на процесе/нити, предузима се из два основна разлога:
  - подизање перформанси
  - раздвајање одговорности
- Као основни мотив за паралелизацију се (исувише) често разматра повећање перформанси
- Раздвајање одговорности је бар једнако значајан мотив као и перформансе, а врло често је и важнији

Универзитет у Београду - Математички факултет

Конкурентно програмирање / Мотивација

Мотивација – подизање перформанси

- Ако је потребно да се неки посао уради над великом количином података, а на располагању је више процесора, онда се ефикасност може вишеструко увећати поделом посла на више токова извршавања (процеса/нити), од којих сваки ради над делом података
  - На пример, ако је потребно да се примени нека локална трансформација слике, онда слика може да се подели на области и обрада сваке од области повери другом процесору
- Таква подела често није сасвим природна и може да значајно подигне сложеност архитектуре
  - У таквим случајевима се често разматра као вид оптимизације

Универзитет у Београду - Математички факултет

Мотивација – подизање перформанси (2)

- Повећање перформанси се помоћу конкурентсности остварује на три основна начина:
  - **смањивање чекања на израчунавање** (*latency reduction*)
    - један посао се дели на више нити
    - убрзава се израчунавање
    - неопходно је паралелизовати алгоритам израчунавања
  - **прикривање чекања** (*latency hiding*)
    - омогућавање да систем ради друге послове док траје обрада
    - алтернатива је употреба неблокирајућих техника
  - **повећавање пропусне моћи** (*throughput increase*)
    - послови се додељују различитим нитима
    - систем уради укупно више посла у јединици времена
    - паралелизује се механизам обраде а не појединачни алгоритми

Мотивација – раздвајање одговорности

- Раздвајање одговорности на нити/процесе омогућава да раздвојимо делове кода који раде целовите делове посла
  - На пример:
    - једна нит комуницира са корисником и прави задатке
    - друга нит (или више њих) асинхроно извршава задатке и чува резултате
    - трећа нит асинхроно приказује докле се стигло са извршавањем
  - Често различити процеси/нити имају улоге клијената или сервера у односу на друге нити
  - Послови се могу додатно раздвојити тако да се извршавају на различитим рачунарима (дистрибуирано извршавање)

Мотивација – раздвајање одговорности (2)

- Иако може да изгледа као подизање сложености, раздвајање одговорности на процесе/нити је
  - често природнији начин решавања проблема и
  - може да има много нижу сложеност него вештачко спајање у један поступак

Мотивација – раздвајање одговорности (3)

- Неки обрасци конкурентног раздвајања одговорности:
  - сигуран интерфејс (адаптер безбедан за нити, монитор-објекти)
    - обезбеђивање приступа објектима
  - предводник и пратиоци
    - послови се стављају у ред послова
    - нека нит ће асинхроно прихватити и урадити посао
  - пул нити
    - посао се додељује некој од нити у пулу
    - ако нема слободне нити, чека се да се ослободи
    - распоређивање посла је синхроно, а извршавање може да буде асинхроно
  - …

Процес

- Процес је инстанца програма која се извршава на рачунарском систему
  - обично се разматра само у контексту рачунарских система који имају могућност извршавања више програма (или инстанци програма) у “исто” време (или стварање довољно добре апстракције таквог извршавања – конкурентно)
  - раније је дефиниција подразумевала да се један процес извршава секвенцијално, али то више није тако
  - процес обухвата
    - код програма који се извршава
    - текуће стање процеса

Стање процеса

- Стање процеса обухвата
  - податке о извршавању
    - стање извршавања (нпр. спреман, ради, чека, стоји,...)
    - бројач инструкција (адреса наредне инструкције)
    - сачуване вредности регистара процесора
  - информације о управљању ресурсима
    - информације о меморији (таблица страница, подаци о алокацији меморије,...)
    - дескриптори датотека
    - остали ресурси, попут У/И захтева и сл.

Промена контекста

- *Промена контекста* (енгл. *context switch*) је промена стања процесора која је неопходна у случају када процесор са извршавања кода једног процеса прелази на извршавање кода другог процеса
  - Стање претходно извршаваног процеса се записује у меморији
  - Стање наредног процеса који ће се извршавати се чита из меморије

Цена процеса

- Процеси су двојако скупи
  - Промена контекста узима значајно време процесора
    - промена контекста се дешава веома често
      - од неколико пута до неколико хиљада пута у секунди
    - велики број процеса може да ослаби перформансе система
  - Подаци о стању процеса су обимни
    - заузумају много меморије
    - при промени контекста се врло често промашују у кешу

Мотивација за увођење нити

- Концепт *нити* извршавања кода се уводи како би се спустила цена промене контекста
  - Једном процесу може да одговара више нити извршавања
  - Ако се већина података води на нивоу процеса и дели међу нитима једног процеса, штеди се на ресурсима и добија на перформансама

Нит

- *Нит извршавања* је компонента процеса која се извршава секвенцијално
  - један процес може имати више нити
    - ако рачунарски систем то подржава
  - сваки процес започиње са једном *главном* нити

Стање процеса и нити

- Подаци који се воде на нивоу процеса (заједнички за све нити тог процеса) обухватају:
  - код програма који се извршава
  - информације о управљању ресурсима
    - информације о меморији (таблица страница, подаци о алокацији меморије,...)
    - дескриптори датотека
    - остали ресурси, попут У/И захтева и сл.
  - стање извршавања процеса
- Подаци о нити (потенцијално другачији за сваку нит) обухватају:
  - стање извршавања нити
  - бројач инструкција
  - сачуване вредности регистара процесора

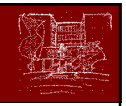
Сличности и разлике

- И процеси и нити могу да се извршавају конкурентно и/или паралелно
  - значајан број проблема при конкурентном извршавању се односи на исти начин и на процесе и на нити
- Само процеси могу да се израчунавају дистрибуирано
  - све нити једног процеса раде у истом адресном простору
- Процеси не деле међу собом ресурсе (бар не непосредно)
  - комуникација између процеса се одвија искључиво кроз посебне механизме за *међупроцесну комуникацију*
  - није потребно експлицитно старање о евентуалном сукобљавању око већине ресурса (нпр. меморија)
    - али јесте за неке дељене ресурсе (нпр. фајлови)
- Нити једног процеса деле све ресурсе тог процеса
  - комуникација између нити се најчешће одвија посредством дељених ресурса
  - неопходно је експлицитно старање о евентуалном сукобљавању око ресурса

Промена контекста нити

- Обично кажемо да је код нити ефикаснија промена контекста...
- ...али морамо да будемо опрезни
  - ако процесор (језгро) прелази са извршавања једне нити на извршавање друге нити истог процеса
    - онда ***јесте*** далеко ефикаснија промена контекста
  - али ако прелази са једне нити на другу нит другог процеса
    - онда се ту мења процес и плаћа се цена промене контекста процеса
- Веома је значајна улога ОС
  - мора да уме да распоређује нити истих процеса на иста језра

C++ и нити



- Велики број библиотека:
  - Стандардна библиотека *C++*
  - *Qt*
  - *Boost*
  - *OpenThreads*
  - *OpenMP*
  - *POSIX Threads*
  - ...

C++ и нити (2)

- Обрадићемо две:
  - *Qt*
    - Прилично добра библиотека солидних могућности
    - Део распрострањеног *Qt* окружења
  - Стандардна библиотека *C++*
    - Уведена у *C++11* и касније унапређивана
    - До данас практично (скоро) универзално расположива

Подршка за нити у *std*

- Основу подршке у *C++ 11* представља класа ***thread***
- Основни методи:
  - конструктор: **thread(fn)**
  - конструктор: **thread(fn, arg1,...)**
  - чекање на завршетак: **void join()**
- Раније је могло да буде неких недоследности имплементација у вези параметара и неких додатних библиотека:
  - **g++ -std=c++11**
  - **g++ -std=c++11 –pthread**
  - **g++ -std=c++0x**

Основне операције са нитима

- Прављење
- Довршавање
- Суспендовање и настављање
- Прекидање
- Чекање

Прављење нити у *std*

- **Прављење**
  - прављење нити на нивоу ОС-а обично подразумева и започињање њеног извршавања
  - прављење објекта нити ***std::thread*** подразумева да ће конструктор објекта направити и нит на нивоу ОС-а
  - нит ће одмах бити направљена и покренута
  - зато не постоји посебан метод за покретање
- Довршавање
- Суспендовање и настављање
- Прекидање
- Чекање

Прављење нити у *std*

- Конструктор објекта класе ***thread***
  - Објекат је средство за приступање нити ОС-а
  - Нит се аутоматски прави и покреће
  - Аргументи конструктора су
    - "главна" функција нити, или нешто што се понаша као функција
      - показивач на функцију, функцијски објекат, ламбда израз
    - опциони аргументи за покретање главне функције нити
- Примери:

```
void fn1() { ... } ...
void fn2( int a, int b, const char* b ) { ... }
...
thread t1(fn1);
thread t2(fn2, 10, 20, "Primer sa parametrima");
thread t3([](){...});
```

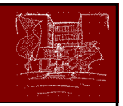

Довршавање нити у *std*

- Прављење
- **Довршавање**
  - Нит се довршава када се заврши извршавање "главне" функције нити
  - Објекат нити се при томе неће аутоматски обрисати
- Суспендовање и настављање
- Прекидање
- Чекање

Суспендовање нити у *std*

- Прављење
- Довршавање
- **Суспендовање и настављање**
  - Нити се не могу прекидати ни настављати од стране других нити
  - Сама нит може да се привремено суспендује препуштајући процесорско време другим нитима (*sleep*, *yield*)
- Прекидање
- Чекање

Суспендовање нити у *std*



- Простор имена ***this_thread*** обухвата операције над текућом нити
- Нит може да привремено суспендује своје извршавање на одређено време или до датог тренутка:

```
template< class Rep, class Period >
void this_thread::sleep_for(
   std::chrono::duration<Rep,Period> sleep_duration )

template< class Clock, class Duration >
void this_thread:: sleep_until(
   const std::chrono::time_point<Clock,Duration>& sleep_time )
```

- Нит може да експлицитно прекине извршавање у додељеном оквиру времена, па да настави са радом када следећи пут добије процесор:

```
void this_thread::yield()
```


Прекидање нити у *std*

- Прављење
- Довршавање
- Суспендовање и настављање
- **Прекидање**
  - једна нит начелно може да прекине извршавање друге нити
  - то је потенцијално веома осетљива операција
    - прекидање нити може да угрози конзистентност података
    - може да има фаталне последице по синхронизацију
  - стандардна библиотека **не нуди** операције за прекидање других нити
- Чекање

Чекање нити у *std*

- Прављење
- Довршавање
- Суспендовање и настављање
- Прекидање
- **Чекање**
  - елементаран вид синхонизовања нити је када једна нит чека да друга заврши са радом
  - остваривање чекања је релативно једноставно, али се препоручује имплементација сложенијих механизама
    - на пример, нит која завршава може да поставља неки дељени сигнал, који се затим може лако проверавати од стране других нити

Чекање нити у *std*

- Чекање да нека нит заврши са радом се изводи позивањем метода:
  ```
  void join()
  ```
- Метод ***join*** завршава рад тек када одговарајућа нит заврши извршавање
- На истој нити не сме да се позове два пута!
- Пример:
  ```
  thread t1([](){ ... });
  t1.join();
  ```


Део интерфејса класе *thread*

- Израчунавање јединственог идентификатора:
  ```
  int get_id() const
  ```
  - на пример, провера да ли тренутно ради управо нит ***t1***:
    ```
    if( t1.get_id() == this_thread::get_id() )...
    ```
- Проверавање да ли може да се чека на нит (извршавање је започето и још није извршен ни метод *join* ни метод *detach* - *ОПАСНО!*):
  ```
  bool joinable() const
  ```
- Чекање на нит:
  ```
  void join()
  ```
- Раздвајање нити од контекста, после тога
  - више не може да се чека на нит
  - нит наставља да се извршава, али објекат је неупотребљив:
    ```
    void detach()
    ```
- Проверавање хардверских могућности (максималан број истовремених нити):
  ```
  static unsigned hardware_concurrency()
  ```


Проблеми конкурентног развоја

- Основни проблеми конкурентног развоја су:
  - подела посла на јединице извршавања
  - дељење ресурса између јединица извршавања
  - комуникација међу јединицама извршавања
  - синхронизација јединица извршавања
- Све се у основи своди на дељење ресурса
- Упознаћемо неколико основних проблема и механизама за њихово решавање

Врсте проблема

- Основне врсте проблема у конкурентним окружењима су:
  - Изгубљене промене (*lost updates*)
  - Приступ непотврђеним изменама (*access to uncommitted data*)
  - Непоновљиво читање (*nonrepeatable read*)
  - Фантомски подаци (*phantom read phenomenon)*
- У трансакционим окружењима су то различити проблеми
- У нетрансакционим окружењима се практично своде на први

Изгубљене промене

- Ако две конкурентне нити (или процеса) читају исти податак и онда га мењају на основу прочитаних вредности, онда може да се изгубити ефекат промене од стране једне од нити:
  - А чита податак
  - Б чита податак
  - А записује податак увећан за 1
  - Б записује податак увећан за 1
- Крајњи исход је да је податак увећан само за 1 а не за 2

Синхронизација

- Проблеми који настају у конкурентним програмима решавају се механизмима за синхронизацију
- Уобичајени механизми за синхронизацију су:
  - мутекси
  - катанци
  - семафори
  - ...
- У трансакционим окружењима се синхронизација решава сложенијим концептом *трансакције*

Мутекси

- Мутекси (енгл. *mutex*) су један од основних начина безбедног дељења података у конкурентним окружењима
- Мутекс има семантику катанца
  - само једанпут може да се закључа
  - ако је мутекс већ закључан, онда ће при наредном покушају закључавања прво морати да се сачека на његово откључавање
- Мутекси се обично имплементирају у оквиру ОС-а тако да операције са мутексима буду атомичне

Мутекси (2)

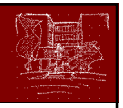
- **ВАЖНО!**
- Мутекси не забрањују приступ други ресурсима сами по себи.
- Закључавање мутекса закључава само мутекс и ништа друго!
- О усклађивању приступања дељеним ресурсима и одговарајућем мутексу морају *доследно* да се старају програмери.

Мутекси у *std*

- Класа ***mutex***
  - објекат мутекса мора да буде доступан свим корисницима за чије дељење се користи
  - закључавање: `lock()`, `try_lock()`
  - откључавање: `unlock()`
- Пример:

```
std::mutex mtx;
void thread_fnc() {
   mtx.lock();
   ...
   mtx.unlock();
}
```


Мутекси у *std (2)*



- Класа ***std::recursive_mutex***
  - Допушта рекурзивно вишеструко закључавање од стране исте нити
    - то иначе није могуће са обичним мутексима
  - Више пута закључан мутекс мора исти број пута да се откључа
- Класа ***std::shared_mutex***
  - Има два нивоа закључавања: ***shared*** и ***exclusive***
  - Као катанац који омогућава дељен и ексклузиван приступ

Мутекси у *std (3)*

- Алтернатива мутексима је шаблон функције ***std::call_once(flag, fn, arg1,...)***
  - Користи помоћни објекат заставице типа ***std::once_flag***
  - У једном тренутку може да буде активан највише један од позива са датом заставицом
  - Аргументи се копирају по вредности

Катанци у *std*

- Семантика катанца – шаблон ***std::lock_guard***
  - Ако се мутекс закључа методом *lock*, а затим дође до изузетка, мутекс може да остане закључан
  - ***std::lock_guard*** се користи тако да се направи аутоматски објекат, који ће се аутоматски обрисати у случају изузетка
  - Мутекс се откључава при брисању катанца, тј. стоји закључан до краја опсега важења катанца.
- Пример:

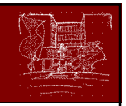
```
std::mutex mtx;
void thread_fnc() {
   ...
   std::lock_guard<std::mutex> lck (mtx);
   ...
}
```

Катанци у *std* (2)

- Новија верзија (*C++17*) – шаблон ***std::scoped_lock***
  - допушта навођење више од једног мутекса одједном
  - закључава их "истовремено"
    - таква је семантика – или успева и сви су закључани, или чека
    - небитан је редослед навођења мутекса
  - препоручује се да се више не користи ***lock_guard*** већ само ***scoped_lock***
- Пример:

```
std::mutex mtx1;
std::mutex mtx2;
void thread_fnc() {
   ...
   std::scoped_lock lck(mtx1, mtx2);
   ...
}
```


Катанци у *std* (3)



- Нешто флексибилније – шаблон ***unique_lock***
  - у основи слично као слично ***lock_guard***
  - али допушта и експлицитно откључавање и закључавање пре краја опсега важења
- Пример:

```
std::mutex mtx;
void thread_fnc() {
   ...
   std::unique_lock<std::mutex> lck (mtx);
   ...
   lck.unlock();
   ...
   lck.lock();
   ...
}
```


Катанци у *std* (4)

- Шаблон функције ***std::lock(lockable)***
  - закључава објекат методом ***lock*** али при томе се стара да не направи мртву петљу (да не покуша вишеструко закључавање ако није допуштено)
  - решава потенцијалан проблем када једна нит покуша да закључа нерекурзиван мутекс који је већ сама закључала
  - има проблем са изузецима, па се препоручује употреба ***std::scoped_lock***

Пример 1: Две једноставне нити користе *cout*

```
#include <iostream>
#include <thread>
using namespace std;

void myThreadFn( int tId )
{
   for( int i=0; i<10; i++ ){
      cout << "Nit " << tId
           << " : " << i << " : ";
      for( unsigned i=0; i<40000; i++ )
         if( i%1000 == 0 )
            cout << tId;
      cout << "\n";
   }
}
```

```
int main(int argc, char **argv)
{
   cout << "Pripremi i kreni..."
        << endl;
   thread t1( myThreadFn, 1 );
   thread t2( myThreadFn, 2 );

   cout << "...sacekaj...\n";
   t1.join();
   t2.join();

   cout << "Kraj." << endl;
   return 0;
}
```

Нити се (као и процеси) извршавају конкурентно – ток извршавања није предвидив

```
Pripremi i kreni...
Nit 1 : 0 : 1Nit 2 : 0 : 21wait...
12121211122121212121212121212121212121212121212121222121212121212121212121212
Nit 1 : 1 : 12121
Nit 2 : 1 : 212121212121212121212121212121212121212121212121212121212121212121212121212121212
Nit 1 : 2 : 12121
Nit 2 : 2 : 212121212121212121212121212121212121212121212121212121212121212121212121212121212
Nit 1 : 3 : 12121
Nit 2 : 31 : 212121212121212121212121212121212121212121212121212121212121212121212121212121212
Nit 1 : 4 : 1212121
Nit 2 : 4 : 212121212121212121212121212121212121212121212121212121212121212121212121212121212
Nit 1 : 5 : 1212121
Nit 2 : 5 : 212121212121212121212121211212121212121212121212121212121212121212121212121212
Nit 1 : 6 : 121212121
Nit 2 : 6 : 212121212121212121212121212121212121212121212121212121212121212121212121212121212
Nit 1 : 7 : 121212121
Nit 2 : 7 : 212121212121212121212122121212121212121212121212121212121212121212121212121212
Nit 1 : 8 : 1212121
Nit 2 : 8 : 212121212121212121212121212121212121212121212121212121221212121212121212121212
Nit 1 : 9 : 12121
Nit 2 : 9 : 212121212121212121212121212121212121212121212121212121212121212121212121212121212
22
Kraj.
```


Пример 2: Две једноставне нити користе *cout*, али користе мутекс за усклађивање

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

mutex ekran;

void myThreadFn( int tId )
{
   for( int i=0; i<10; i++ ){
      ekran.lock();
      cout << "Nit " << tId
           << " : " << i << " : ";
      for( unsigned i=0; i<40000; i++ )
         if( i%1000 == 0 )
            cout << tId;
      cout << "\n";
      ekran.unlock();
   }
}
```

```
int main(int argc, char **argv)
{
   cout << "Pripremi i kreni..."
        << endl;
   thread t1( myThreadFn, 1 );
   thread t2( myThreadFn, 2 );

   ekran.lock();
   cout << "...sacekaj...\n";
   ekran.unlock();

   t1.join();
   t2.join();

   cout << "Kraj." << endl;
   return 0;
}
```


Пример 3: Као претходни пример, али се користи сигурно закључавање

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

mutex ekran;

void myThreadFn( int tId )
{
   for( int i=0; i<10; i++ ){
      lock_guard<mutex> katanac(ekran);
      cout << "Nit " << tId
           << " : " << i << " : ";
      for( unsigned i=0; i<40000; i++ )
         if( i%1000 == 0 )
            cout << tId;
      cout << "\n";
   }
}
```

```
int main(int argc, char **argv)
{
   cout << "Pripremi i kreni..."
        << endl;
   thread t1( myThreadFn, 1 );
   thread t2( myThreadFn, 2 );

   {
   lock_guard<mutex> kat(ekran);
   cout << "...sacekaj...\n";
   }

   t1.join();
   t2.join();

   cout << "Kraj." << endl;
   return 0;
}
```


Пример 4: Програм са 2 нити које деле глобални податак

```
#include <iostream>
#include <thread>
#include <vector>
using namespace std;

int globalData = 0;

void myThreadFn( int tId )
{
   for( int i=0; i<10; i++ ){
      int x = globalData;
      cout << "Nit " << tId
           << " : " << i << endl;
      x++;
      globalData = x;
   }
}
```

```
int main(int argc, char **argv)
{
   int n = 100;
   vector<thread> threads;

   cout << "Pripremi i kreni..."
        << endl;
   for( int i=0; i<n; i++ )
      threads.emplace_back(
         myThreadFn, i );

   cout << "...sacekaj...\n";

   for( thread& t: threads )
      t.join();

   cout << "Kraj." << endl;
   cout << "globalData = "
        << globalData << endl;
   return 0;
}
```

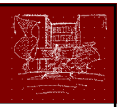
Пример 5: Као претходни пример, али закључавамо податак

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
using namespace std;

int globalData = 0;
mutex globalDataMutex;

void myThreadFn( int tId )
{
   for( int i=0; i<10; i++ ){
      lock_guard<mutex>
         gdmLock(globalDataMutex);
      int x = globalData;
      cout << "Nit " << tId
           << " : " << i << endl;
      x++;
      globalData = x;
   }
}
```

```
int main(int argc, char **argv)
{
   int n = 100;
   vector<thread> threads;

   cout << "Pripremi i kreni..."
        << endl;
   for( int i=0; i<n; i++ )
      threads.emplace_back(
         myThreadFn, i );

   cout << "...sacekaj...\n";

   for( thread& t: threads )
      t.join();

   cout << "Kraj." << endl;
   cout << "globalData = "
        << globalData << endl;
   return 0;
}
```


Пример 6: Програм који обрађује задатке у више нити

- Главни програм:
  - прави празну колекцију задатака
  - покреће више нити за извршавање задатака
  - прихвата податке са улаза и додаје у колекцију задатака
- Свака од нити за извршавање задатака
  - узима нови задатак из колекције
  - извршава га
  - ако тренутно нема задатака, привремено се успава



Пример 6: Програм који обрађује задатке у више нити

- Задаци су једноставни:
  - задатак је описан природним бројем
  - потребно је исписати бројеве од 1 до датог броја
- Када се унесе број који је мањи од нуле, програм стаје

Пример 6: Програм који обрађује задатке у више нити – главни програм

```
void postavljanjeZadataka( Zadaci& zadaci )
{
   while(true){
      cout << "Unesi pozitivan ceo broj: ";
      int n;
      cin >> n;
      if( n<0 )
         cout << "Greska!" << endl;
      else{
         zadaci.dodajZadatak(n);
         if( n==0 )
            break;
      }
   }
}
```

```
int main(int argc, char **argv)
{
   int n = 5;
   vector<thread> threads;
   Zadaci zadaci;

   cout << "Pokretanje niti..." << endl;
   for( int i=0; i<n; i++ )
      threads.emplace_back(
         izvrsavanjeZadataka, ref(zadaci), i );

   postavljanjeZadataka(zadaci);
   cout << "Kraj." << endl;
   for( thread& t: threads )
      t.join();

   cout << "end..." << endl;
   return 0;
}
```

Пример 6: Програм који обрађује задатке у више нити – извршавање задатака

```
void izvrsavanjeZadataka( Zadaci& zadaci, int _ID )
{
   while(true){
      int zadatak = zadaci.uzmiZadatak();
      if( !zadatak )
         break;
      if( zadatak > 0 ){
         for( int i=1; i<=zadatak; i++ ){
            cout << "Nit " << _ID << " : " << i << endl;
            this_thread::sleep_for( chrono::milliseconds( 200 ));
         }
      }
      else
         this_thread::sleep_for( chrono::milliseconds( 200 ));
   }
}
```

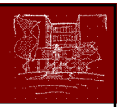

Пример 6: Програм који обрађује задатке у више нити – задаци

```
class Zadaci {
 public:
   Zadaci() {}

   void dodajZadatak( int n ) {
      lock_guard<mutex> lock(zadaciMutex);
      zadaci.push( n );
   }

   int uzmiZadatak() {
      lock_guard<mutex> lock(zadaciMutex);
      if( zadaci.empty() )
         return -1;
      // 0 koristimo kao oznaku za kraj,
      // ostavljamo je u redu poslova da bi i druge niti zavrsile sa radom
      int n = zadaci.front();
      if( n>0 )
         zadaci.pop();
      return n;
   }

 private:
   mutex zadaciMutex;
   queue<int> zadaci;
};
```


Имплицитно чекање на резултат

- Стандардна библиотека има механизме за асинхроно (одложено или паралелно) позивање и чекање на резултат:
  - за позивање се користи функција ***async***
  - за чекање резултата се користи шаблон ***future***

Имплицитно чекање на резултат (2)



- Функција ***std::async***
  - позива асихнхроно дату функцију са датим аргументима
  - прикрива механизам прављења нити
  - враћа као резултат објекат ***std::future***
- Има неколико верзија, у основи:
  - ***future<resultType> async(fn, args...)***
    - позива се дата функција са аргументима
  - ***future<resultType> async(policy, fn, args...)***
    - при позивању се примењује дата политика
      - ***std::launch::async*** – прави се нова нит и покреће се асинхроно
      - ***std::launch::deffered*** – посао се одлаже и покренуће се у истој нити (одложено, лењо) тек када се први пут затражи његов резултат
      - ако се наведу оба (тј. бинарна дисјункција, што је и подразумевано), онда имплементација бира начин извршавања

Имплицитно чекање на резултат (3)

- Шаблон ***std::future***
  - омогућава чекање на резултат до завршетка израчунавања
  - конструктор прави нит и покреће израчунавање
  - методи **wait()**, **wait_for(...)**, **wait_until(...)** чекају на завршетак рачунања
  - метод **get()** чека на завршетак рачунања и враћа резултат
    - сме да се позове највише једанпут
    - имплицитно позива **wait()**
- Обједињује чекање да нит заврши рад и преношење (записивање и читање) резултата
- У већини случајева може да буде једноставније да се користе ***async*** и ***future*** него да се експлицитно користе објекти ***thread***

Имплицитно чекање – пример

- Пример:

```
int fnc() { return 10; }
...
void prg(){
  ...
  std::future<int> primer = std::async( fnc );
  ...
  cout << primer.get() << endl;
}
```


Пример 8: Употреба шаблона *future*

```
#include <iostream>
#include <thread>
#include <future>

using namespace std;

int sumaDo( int tId, int n )
{
  int s = 0;
  for( int i=1; i<=n; i++ ){
    if( i % 1000 == 0 ){
      cout << "Nit " << tId
           <<" : i = " << i << endl;
      this_thread::sleep_for(
        chrono::milliseconds(200));
    }
    s += i;
  }
  cout << "Nit " << tId
       << " : suma 1 do " << n
       << " = " << s << endl;
  return s;
}
```

```
...

int main(int argc, char **argv)
{
  std::future<int> s1 = std::async(
    std::launch::async,
    sumaDo, 1, 10000 );

  std::future<int> s2 = std::async(
    std::launch::async,
    sumaDo, 2, 10000 );

  std::future<int> s3 = std::async(
    std::launch::async,
    sumaDo, 3, 10000 );

  cout << "Rezultat = "
    << s3.get() + s2.get() + s1.get()
    << endl;

  return 0;
}
```


Део интерфејса класе *future*

- **`T get()`**
  - Чека да стање објекта постане довршено и затим га чита
  - Не сме да се употреби више пута
- **`bool valid() const`**
  - Проверава да ли је стање објекта исправно
    - тј. да ли на њему сме да се позове метод **`get()`**
  - Објекат је исправан ако још ниједанпут није позван метод **`get()`**
  - Не зависи од тога да ли је израчунат или не
- **`void wait() const`**
  - Чека да стање објекта постане довршено

Део интерфејса класе *future* (2)

- `future_status wait_for( const chrono::duration<Rep,Period>&) const`
  - Чека најдуже дати период да стање објекта постане довршено
  - Резултат је статус, који може да буде:
    - **future_status::ready** - израчунавање је довршено и објекат може да се користи
    - **future_status::timeout** - истекло је време, а објекат још није довршен
    - **future_status::deferred** - израчунавање није ни почело (нпр. асинхроно одложено)
- `future_status wait_for( const chrono::time_point<Clock,Duration>&) const`
  - Чека најдуже до датог тренутка да стање објекта постане довршено
- `shared_future<T> share()`
  - Прави одговарајући дељени објекат (оригинал постаје неисправан)
  - Дељени објекат ***shared_future*** је по свему другом сличан објекту ***future***
  - Али може да се користи (тј. чита методом ***get***) више пута (и у више нити)

Чекање на међурезултат

- Употреба шаблона ***future*** је ограничена на резултат функције
- Ако је потребно да се чека на један или више међурезултата, чије се вредности експлицитно постављају у другој нити или у некој асинхроној функцији, онда се користи шаблон ***promise***
- Шаблон ***promise***
  - представља објекат међурезултата, чија вредност се чека
    - има семантику поруке коју једна нит шаље другој
  - метод ***set_value(T)*** поставља вредност
  - метод ***set_value_at_thread_exit(T)*** поставља вредност али допушта њену употребу тек по завршетку нити
    - тј. задужује нит да дату вредност постави при завршетку
  - метод ***set_exception(exc)*** поставља изузетак
    - када се приступи објекту наступиће дати изузетак
  - метод ***get_future()*** враћа одговарајући објекат класе ***future***
    - сме да се употреби само једном

Чекање на међурезултат (2)

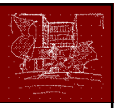
- Пример:

```
std::promise<int> rezultat;

void fnc_nit1() {
  ...
  rezultat.set_value(1);
  ...
  rezultat.set_exception(
       std::make_exception_ptr(
           std::runtime_error("... ")));
  ... наставак рада, нпр. деиницијализација ...
}

int main() {
  future<int> rezultat_obj = rezultat.get_future();
  std::thread t(nit1);
  ...
  rezultat.get();
  ...
}
```


Атомичност података

- Податак је ***атомичан*** ако се све елементарне операције над податком одвијају атомично
  - тј. не могу да буду прекинуте током извршавања, па тиме ни небезбедне по нити
- ***Атомичност података*** је веома важан аспект конкурентног програмирања
- Ако знамо да је нека операција атомична, онда не морамо да је *штитимо* синхронизацијом
  - а ако не знамо, онда морамо...
- Атомичност операција може да представља брже и боље решење од синхронизације

Атомичност података (2)



- Када напишемо једноставне операције, очекујемо да су атомичне:

```
a = b
a++
a+=5
...
```

- Међутим, то није увек случај
  - зависи од много фактора, укључујући и архитектуру процесора

Атомичност података (3)

- У случају фамилија процесора x86, x64 атомичност је начелно обезбеђена на подацима величине 1, 2, 4, 8 бајтова
  - 32-битни процесори не гарантују атомичност операција над 8 бајтова
- У пракси то не мора увек да буде случај
  - Ако неки процесор увек приступа подацима фиксне величине (нпр. 32 бита), а више мањих података је спаковано у тих 32 бита (нпр. више логичких или знаковних података)
  - Тада приступање подацима може да захтева више машинских инструкција
  - Нит може да буде прекинута након што је прочитан или записан само део податка

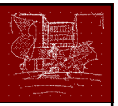
C++ и атомичност података

- Шаблон класе `std::atomic` обезбеђује атомичност основних операција помоћу метода:
  - `store` – записивање податка
    - може да се користи и оператор додељивања
  - `load` – читање података
    - може да се користи и конверзија у основни тип
  - `exchange` – чита стару и уписује нову вредност
    - атомично читање и писање
  - `fetch_add`, `fetch_sub`, `fetch_and`, `fetch_or`, `fetch_xor`
    - извршава операцију и враћа претходни садржај
  - `operator++`, `operator--`, `operator+=`, `operator-=`, `operator&=`, `operator|=`, `operator ^=`
    - уобичајене операције

C++ и атомичност података (2)

- На пример:
  - `atomic<int> deljeniPodatak;`
  - `...`
  - `deljeniPodatak.store(3);`
  - `deljeniPodatak = 3;`
  - `deljeniPodatak += 3;`
  - `...`

C++ и атомичност података (3)



- Постоје и шаблони функција који могу да се употребљавају на сличан начин, али се препоручује употреба шаблона класе **std::atomic**
- Примери функција:
  - atomic_init
  - atomic_store
  - atomic_load
  - atomic_exchange
  - atomic_fetch_add
  - ...

C++ и атомичност података (4)

- Шаблон класе **std::atomic** је предвиђен за вредности које могу да буду атомичне (али не морају), као што су прости типови података
- За сложене типове је обезбеђен шаблон класе **std::atomic_ref**, који обезбеђује атомичне операције на сложенијим објектима
  - Подржан је скоро исти скуп операција, али је имплементација сложенија и мање ефикасна
  - Не треба користити **std::atomic_ref** на местима где може да се користи **std::atomic**

C++ и атомичност података (5)

- Имплементација се стара да операције буду извршене на атомичан начин
- Библиотека је специфична зато што је имплементирана делимично кроз кодирање, а делимично кроз директиве преводиоцу
- Ради се о веома сложеним концептима, који у првим имплементацијама нису оптимално изведени, али је мануелно кодирање веома сложено и не препоручује се
  - У неким случајевима са интензивном применом атомичних операција се испоставља да је ефикасније направити мануелну синхронизацију нити
  - Ако операција не може да се преведе у атомичну машинску инструкцију, онда се своди на примену мутекса, па више таквих операција може да доведе до вишестуких закључавања и откључавања мутекса
- Препоручује се употреба операција `store` и `load`, а не оператора =, ради уочљивости операција на дељеним подацима у коду, чак и онда када смо сигурни да ће операције бити атомичне и без тога

Подршка за нити у оквиру *Qt*

- Основу подршке за нити представља класа ***QThread***
- Основни методи:
  - **bool isFinished()** – проверава да ли је завршила са радом
  - **bool isRunning()** – проверава да ли се нит извршава
  - **bool wait(unsigned long time_msec = ULONG_MAX)** – блокира текућу нит док (1) нит чији метод је позван не заврши са радом или (2) не истекне наведено време у *ms*
- Слотови (могу се користити као методи):
  - **void start()** – покретање нити
  - **void quit()** – саопштава нити да је потребно да прекине рад
  - **void terminate()** – прекида нит (у складу са политикама ОС-а)
- Сигнали:
  - **void finished()** – нит је завршила извршавање
  - **void started()** – нит је започела извршавање
  - **void terminated()** – нит је прекинута

Подршка за нити у оквиру *Qt* (2)

- Нова класа нити наслеђује класу ***QThread***
  - тело нити се имплементира као заштићен метод ***void run()***

```
#include <QThread>
class MyThread : public QThread {
  Q_OBJECT
  public:
    MyThread(...);
  protected:
    void run();
};
```


Пример 1: Две једноставне нити користе *cout*

MyThread.h

```
#ifndef MYTHREAD_H
#define MYTHREAD_H
#include <QThread>

class MyThread : public QThread
{
Q_OBJECT
public:
    MyThread( int id )
        : _ID(id)
    {}
protected:
    void run();
    int _ID;
};
#endif // MYTHREAD_H
```

MyThread.cpp

```
#include <iostream>
#include "MyThread.h"
using namespace std;

void MyThread::run()
{
    for( int i=0; i<10; i++ )
        cout << "Nit " << _ID
             << " : " << i
             << endl;
}
```

main.cpp

```
#include <iostream>
using namespace std;

#include "MyThread.h"

#ifdef QT_NO_CONCURRENT
    !!! Nije podrzano !!!
#endif

int main(int argc, char **argv)
{
    cout << "prepare..." << endl;
    MyThread t1(1);
    MyThread t2(2);

    cout << "run..." << endl;
    t1.start();
    t2.start();

    cout << "wait..." << endl;
    t1.wait();
    t2.wait();

    cout << "end..." << endl;
    cin.get();

    return 0;
}
```


Нити се (као и процеси) извршавају конкурентно – ток извршавања није предвидив

Извршавање 1:

```
prepare...
run...
wait...
Nit Nit 2 : 1 : 0
0
Nit 2Nit  : 1
1 : Nit 12
 : 2
Nit 1Nit  : 2 : 23

Nit 2Nit  : 14
 : Nit 3
2 : 5N
it 1Nit 2 :  : 46
N
it 2 : Nit 71
 : 5N
it Nit 2 : 18
 : Nit 62
 : 9
Nit 1 : 7
Nit 1 : 8
Nit 1 : 9
end...
```

Извршавање 2:

```
prepare...
run...
wait...
Nit 2 : 0
Nit 2 : 1
Nit 2 : 2
Nit 2 : 3
Nit 2 : 4
Nit 2 : 5
Nit 2 : 6
Nit 2 : 7
Nit 2 : 8
Nit 1Nit  : 20
 : Nit 9
1 : 1
Nit 1 : 2
Nit 1 : 3
Nit 1 : 4
Nit 1 : 5
Nit 1 : 6
Nit 1 : 7
Nit 1 : 8
Nit 1 : 9
end...
```


Пример 2: Програм са 2 нити које деле глобални податак

MyThread.h

Исто као у претходном примеру

MyThread.cpp

```
...



extern int globalData;

void MyThread::run()
{
    for( int i=0; i<10; i++ ){
        int x = globalData;
        cout << "Nit " << _ID
             << " : " << i
             << endl;
        x++;
        globalData = x;
    }
}
```

main.cpp

```
...

int globalData = 0;

int main(int argc, char **argv)
{
    ...

    cout << "end..." << endl;
    cout << "globalData = "
         << globalData << endl;

    cin.get();
    return 0;
}
```

Пример 2а: Програм са више нити које деле глобални податак

MyThread.h
Исто као у претходном примеру

MyThread.cpp
Исто као у претходном примеру

main.cpp

```
...
int main(int argc, char **argv)
{
   int n = 10;
   MyThread** threads = new MyThread*[n];

   cout << "prepare..." << endl;
   for( int i=0; i<n; i++ )
     threads[i] = new MyThread(i);

   cout << "run..." << endl;
   for( int i=0; i<n; i++ )
     threads[i]->start();

   cout << "wait..." << endl;
   for( int i=0; i<n; i++ )
     threads[i]->wait();

   cout << "end..." << endl;
   for( int i=0; i<n; i++ )
     delete threads[i];
   delete [] threads;

   cout << "globalData = "
        << globalData << endl;
   cin.get();
   return 0;
}
```


Пример 2б: Програм са 2 нити које деле глобални податак, део 2

MyThread.h
Исто као у претходном примеру

MyThread.cpp
Ако изменимо редослед корака, промениће се резултат
...

```
extern int globalData;

void MyThread::run()
{
    for( int i=0; i<10; i++ ){
        int x = globalData;
        x++;
        globalData = x;
        cout << "Nit " << _ID
             << " : " << i
             << endl;
    }
}
```

main.cpp

```
...

int globalData = 0;

int main(int argc, char **argv)
{
    ...

    cout << "end..." << endl;
    cout << "globalData = "
         << globalData << endl;

    cin.get();
    return 0;
}
```


Основне операције са нитима

- Прављење
- Довршавање
- Суспендовање и настављање
- Прекидање
- Чекање

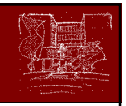
Основне операције са нитима

- **Прављење**
  - прављење нити на нивоу ОС-а обично подразумева и започињање њеног извршавања
  - прављење објекта нити *QThread* не подразумева и прављење нити на нивоу ОС-а
  - нит ће бити направљена на нивоу ОС-а тек позивањем метода *start()*
- Довршавање
- Суспендовање и настављање
- Прекидање
- Чекање

Прављење нити у *Qt*

- Објекат класе ***QThread*** представља помоћно средство за руковање нитима ОС-а
- Конструкција објекта **не обухвата** прављење нити
- Нова нит се заиста прави (на нивоу ОС-а) када се позове метод ***void start()*** објекта класе ***QThread***

Основне операције са нитима



- Прављење
- **Довршавање**
  - када нит доврши своје извршавање, ослобађа се већина ресурса везаних за нит
  - остаје само коначан статус нити (резултат извршавања, слично функцији *main*)
- Суспендовање и настављање
- Прекидање
- Чекање

Довршавање нити у *Qt*

- Нит се довршава када се заврши извршавање метода ***void run()*** објекта класе ***QThread***
- Динамички направљен објекат ће бити аутоматски уклоњен ако му се при конструкцији додели родитељски објекат

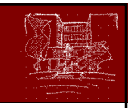
Основне операције са нитима

- Прављење
- Довршавање
- **Суспендовање и настављање**
  - неки ОС омогућавају да се нит привремено суспендује и да се касније њено извршавање настави
    - *Windows, Solaris*
  - код ОС који не подржавају суспендовање, одговарајуће понашање се може остварити само сложенијим механизмима за синхронизацију
    - разлози су вишеструки, али се сумирају у потенцијално неконтролисаном држању закључаних ресурса од стране суспендоване нити
    - *POSIX*
- Прекидање
- Чекање

Суспендовање нити у *Qt*

- Суспендовање нити није могуће

Основне операције са нитима



- Прављење
- Довршавање
- Суспендовање и настављање
- **Прекидање**
  - једна нит може да прекине извршавање друге нити
  - иако се остварује наизглед једноставно (позивом једне фнкције ОС-а) ово је веома осетљива операција
  - прекидање нити прети да угрози конзистентност дељених података
- Чекање

Прекидање нити у *Qt*

- Нит може да се прекине позивањем метода ***void terminate()***
- Нит не мора бити прекинута током извршавања овог метода
  - када ће стварно нит бити прекинута зависи од одговарајуће политике ОС-а
  - ако је потребно да се нит поуздано заврши, након позивања овог метода се може сачекати да се нит заврши

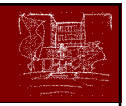
Основне операције са нитима

- Прављење
- Довршавање
- Суспендовање и настављање
- Прекидање
- **Чекање**
  - елементаран вид синхонизовања нити је када једна нит чека да друга заврши са радом
  - остваривање чекања је релативно једноставно, али се препоручује имплементација сложенијих механизама
    - на пример, нит која завршава може да поставља неки дељени сигнал, који се затим може лако проверавати од стране других нити

Чекање нити у *Qt*

- Нит може да се чека позивањем метода
  bool wait(unsigned long time = ULONG_MAX)
- Ако се не наведе аргумент (или се наведе *ULONG_MAX*), метод чека док се нит не заврши
- Иначе, метод чека на завршавање нити највише ***time*** милисекунди
- Враћа ***true*** ако нит више није активна (или је завршена или није ни започела) и ***false*** ако је нит још активна

Мутекси у *Qt*



- Мутекси су имплементирани класом ***QMutex***
- Основни методи:
  - ***void lock()*** закључава мутекс
    - ако је већ закључан, чека да се откључа па га закључава
    - у сваком случају, враћа се тек када је успешно закључан
  - ***void unlock()*** откључава мутекс
    - ако је закључан од стране исте нити, откључа га
    - ако је закључан од стране друге нити, производи грешку
    - ако није закључан понашање је недефинисано
  - ***bool tryLock()*** покушава да закључа мутекс
    - ако је мутекс откључан, закључава га и враћа *true*
    - ако је већ закључан, не ради ништа и одмах враћа *false*
    - опциони аргумент је макс. трајање чекања у *ms*
  - конструктор **QMutex (RecursionMode mode = NonRecursive)**
    - опциони аргумент може да буде **Qmutex::Recursive**
      - тада једна нит може да закључа мутекс више пута узастопно
      - мора да га откључа онолико пута колико га је пута закључала

Употреба мутекса

- Уобичајено је да се једним мутексом обезбеђује један податак или скуп података који у једном тренутку сме да употребљава само једна нит
  - На почетку атомичног поступка мутекс се закључава
  - На крају атомичног поступка се мутекс откључава
- Веома је важно да се сваки закључани мутекс откључа
- Ако један исти мутекс закључава различите податке, онда се потенцијално спушта ниво конкурентности
- Ако више мутекса закључава један исти података, отвара се простор за озбиљне превиде у управљању тим ресурсом

Пример 3: Употреба мутекса

MyThread.h
Исто као у претходном примеру

MyThread.cpp

```
#include <iostream>
#include <QMutex>
#include "MyThread.h"

using namespace std;

QMutex globalDataMutex;
extern int globalData;

void MyThread::run()
{
   for( int i=0; i<10; i++ ){
      globalDataMutex.lock();
      int x = globalData;
      cout << "Nit " << _ID
           << " : " << i << endl;
      x++;
      globalData = x;
      globalDataMutex.unlock();
   }
}
```

main.cpp
Исто као у претходном примеру

Класа *QMutexLocker*

- Класа ***QMutexLocker*** поједностављује рад са мутексима и чини га исправним и у случају изузетака
  - употребљава се искључиво за аутоматске објекте
  - објекат се прави за дати конкретан мутекс
    - конструктор ***QMutexLocker(QMutex*)***
    - чином прављења објекта мутекс се закључава
  - аутоматски објекат се аутоматски уклања при изласку из блока
    - при уклањању се откључава катанац
    - ради исправно чак и у контексту изузетака
- Добар начин да се предупреде грешке са пропуштањем откључавања

Пример 4: Употреба мутекса посредством помоћног објекта класе *QMutexLocker*

MyThread.h
Исто као у претходном примеру

MyThread.cpp

```
#include <iostream>
#include <QMutex>
#include <QMutexLocker>
#include "MyThread.h"

using namespace std;

QMutex globalDataMutex;
extern int globalData;

void MyThread::run()
{
   for( int i=0; i<10; i++ ){
      QMutexLocker lock(&globalDataMutex);
      int x = globalData;
      cout << "Nit " << _ID
           << " : " << i << endl;
      x++;
      globalData = x;
   }
}
```

main.cpp
Исто као у претходном примеру

Катанци

- Мутекси се понашају као специјалан случај катанаца
  - имају само ексклузиван режим приступа
- Апстракција катанца омогућава различите приступе под различитим условима
- Уобичајено
  - ако једна нит чита податке, тада и друге нити могу читати податке
    - тзв. дељени катанац
  - ако једна нит мења податке, нико други ни на који начин не сме користити податке

Катанци у *Qt*

- Катанци су имплементирани класом ***QReadWriteLock***
- Основни методи:
  - ***void lockForRead()***
    - поставља катанац за читање
    - ако је закључан за писање, метод чека да се катанац за писање откључа
  - ***void lockForWrite()***
    - поставља катанац за писање
    - ако је закључан (било како), метод чека да се сви катанци откључају
  - ***void unlock()***
    - откључава катанац
  - ***bool tryLockForRead(), bool tryLockForWrite()***
    - покушавају да закључају катанац
    - опциони аргумент је макс. трајање чекања у *ms*
  - конструктор ***QReadWriteLock(RecursionMode mode = NonRecursive)***
    - опциони аргумент може да буде ***QReadWriteLock*::Recursive**

Класе *QReadLocker, QWriteLocker*

- Слично класи ***QMutexLocker*** поједностављују рад са катанцима
- Добар начин да се предупреде грешке са пропуштањем откључавања

Пример 5: Употреба катанаца посредством помоћних објеката

MyThread.h

Исто као у претходном примеру

MyThread.cpp

```
#include <iostream>
#include <QReadWriteLock>
#include "MyThread.h"
using namespace std;

QReadWriteLock globalDataLock;
extern int globalData;

int readGlobalData()
{
    QReadLocker lock(&globalDataLock);
    return globalData;
}

void MyThread::run()
{
   for( int i=0; i<10; i++ ){
      msleep(20);
      QWriteLocker lock(&globalDataLock);
      int x = globalData;
      cout << "Nit " << ID << " : "
           << i << endl;
      x++;
      globalData = x;
   }
}
```

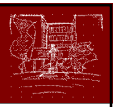
main.cpp

```
...
int globalData = 0;
int readGlobalData();

int main(int argc, char **argv)
{
...
  cout << "wait..." << endl;
  for( int i=0; i<n; i++ ){
    while(1){
      cout << "Result = "
           << readGlobalData()
           << endl;
      if( threads[i]->wait(50) )
        break;
    }
  }
...
}
```


Семафори

- Један од основних механизама за синхронизацију и деобу ресурса представљају *семафори*
- Семафори омогућавају да се неки бројач контролисано и безбедно повећава, смањује и проверава
- Имају семантику бројача слободних ресурса

Семафори у *Qt*



- Семафори су имплементирани класом ***QSemaphore***
- Основни методи:
  - ***int available()***
    - враћа тренутну вредност семафора
  - ***void acquire(int n = 1)***
    - смањује семафор за *n*
    - ако је *n* > *available()*, чека да постане *n* <= *available()*
  - ***void release(int n = 1)***
    - увећава семафор за *n*
    - метод се употребљава и за "прављење нових ресурса"
  - ***bool tryAcquire(int n = 1)***
    - покушава да смањи семафор за *n*
    - ако успе враћа *true*, а иначе *false*
  - ***bool tryAcquire(int n, int timeout)***
    - покушава да смањи семафор за *n* и чека најдуже *timeout ms*
    - ако успе враћа *true*, а иначе *false*
  - конструктор ***QSemaphore(int n=0)***
    - прави семафор и иницијализује бројач датом вредношћу

Пример 6: Употреба семафора за проверавање да ли су сви процеси завршили посао

MyThread.h

Исто као у претходном примеру

MyThread.cpp

```
#include <iostream>
#include <QReadWriteLock>
#include <QSemaphore>
#include "MyThread.h"
using namespace std;

QReadWriteLock globalDataLock;
extern int globalData;
extern QSemaphore threadSem;

int readGlobalData()
{...}

void MyThread::run()
{
   for( int i=0; i<10; i++ ){
      msleep(20);
      QWriteLocker lock(&globalDataLock);
      int x = globalData;
      cout << "Nit " << ID << " : "
           << i << endl;
      x++;
      globalData = x;
   }
   threadSem.release();
}
```

main.cpp

```
...
QSemaphore threadSem;
int globalData = 0;
int readGlobalData();

int main(int argc, char **argv)
{
...
  cout << "wait..." << endl;
  forever{
    if( threadSem.available() == n )
      break;
    cout << "Result = "
         << readGlobalData() << endl;
    }
...
}
```


Пример програма

- Функција *main*
  - прави и покреће 10 нити које извршавају задатке
  - корисник уноси задатке (0 за крај)
  - чека да све нити заврше извршавање
- Класа *Zadaci*
  - ред задатака
    - задатак је број *n*
    - потребно је исписати бројеве од 1 до *n*
    - задатак 0 означава крај
  - безбедна у вишенитном окружењу
- Класа *MyThread*
  - нит која извршава задатке
  - безбедан бројач колико је нити довршило посао

Пример 7: Програм који обрађује задатке у више нити

Zadaci.h

```
#include <queue>
#include <QReadWriteLock>

class Zadaci
{
public:
    Zadaci()
    {}

    void dodajZadatak( int n )
    {
        QWriteLocker lock(&zadaciLock);
        zadaci.push( n );
    }

    int uzmiZadatak()
    {
        QWriteLocker lock(&zadaciLock);
        if( zadaci.empty() )
            return -1;
        // 0 koristimo kao oznaku za kraj,
        // ostavljamo je da bi i druge niti zavrsile sa radom
        int n = zadaci.front();
        if( n>0 )
            zadaci.pop();
        return n;
    }

private:
    QReadWriteLock  zadaciLock;
    std::queue<int> zadaci;
};
```


Пример 7: Програм који обрађује задатке у више нити

MyThread.h

```
#include <QThread>
#include <QSemaphore>
#include "Zadaci.h"

class MyThread : public QThread
{
Q_OBJECT
public:
   MyThread( int id, Zadaci& z )
      : _ID(id), _zadaci(z)
   {}

   static int Finished()
   {
      return _finished.available();
   }

protected:
   void run();
   int _ID;
   Zadaci& _zadaci;
   static QSemaphore _finished;
};
```

MyThread.cpp

```
#include "MyThread.h"
#include <iostream>

using namespace std;

QSemaphore MyThread::_finished;

void MyThread::run()
{
   forever{
      int zadatak = _zadaci.uzmiZadatak();
      if( !zadatak )
         break;
      if( zadatak > 0 ){
         for( int i=1; i<=zadatak; i++ ){
            cout << "Nit " << _ID
                 << " : " << i << endl;
            msleep(100);
            }
      }else{
         msleep(100);
      }
   }
   _finished.release();
}
```

Пример 7: Програм који обрађује задатке у више нити

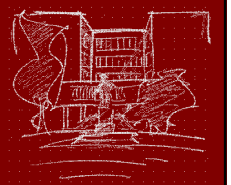
main.cpp

```
#include <QSemaphore>
#include <iostream>
using namespace std;

#include "MyThread.h"

#ifdef QT_NO_CONCURRENT
    !!! Nije podrzano !!!
#endif

int main(int argc, char **argv)
{
  Zadaci zadaci;

  int n = 10;
  MyThread** threads = new MyThread*[n];

  cout << "pokretanje niti..."
       << endl;
  for( int i=0; i<n; i++ ){
    threads[i] = new MyThread(i,zadaci);
    threads[i]->start();
  }
  ...
  ...
  // zadaci
  forever{
    cout << "Unesi pozitivan "
            "ceo broj: ";
    int n;
    cin >> n;
    if( n<0 )
      cout << "Greska!" << endl;
    else{
      zadaci.dodajZadatak(n);
      if( n==0 )
        break;
    }
  }

  cout << "wait..." << endl;
  forever
    if( MyThread::Finished() == n )
      break;

  cout << "end..." << endl;
  for( int i=0; i<n; i++ )
    delete threads[i];
  delete [] threads;

  return 0;
}
```


Литература за тему

- ***Qt Reference Documentation***, *http://doc.qt.io/*
- *http://www.cplusplus.com*
- *http://www.cppreference.com*
- Clay Breshears, **The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications**, *O'Reilly Media (2009)*

Хвала на пажњи!



МАТФ
Универзитет у Београду
Математички факултет