მონაცემების გაზიარება დინებებს შორის

განხილული საკითხები:

- რა სირთულეები ახლავს დინებებს შორის მონაცემთა გაზიარებას?
- რბოლის ვითარება (race conditions). მასთან გამკლავება >>>
- საზიარო მონაცემების დაცვა მუტექსით (mutex) >>>
- მუტექსის გამოყენება C++ -ში >>>
- კოდის სტრუქტურირება საზიარო მონაცემების დაცვის მიზნით >>>
- ინტერფეისებისთვის დამახასიათებელი რბოლის ვითარებების გამოკვლევა >>>
- ჩიხი: სირთულე და გამოსავალი >>>
- გზამკვლევები ჩიხის ასარიდებლად >>>
- მოქნილი ჩაკეტვა std::unique_lock -ით >>>
- ჩაკეტვის მოქმედების სივრცე >>>
- საზიარო მონაცემების დაცვა კერძო შემთხვევებში: ინციალიზაციის პროცესის დაცვა >>>
- საზიარო მონაცემების დაცვა კერძო შემთხვევებში: იშვიათად განახლებადი მონაცემების დაცვა >>>

სავარჯიშოები >>>

რა სირთულეები ახლავს დინებებს შორის მონაცემთა გაზიარებას?

დინებებს შორის მონაცემთა გაზიარებასთან დაკავშირებული გართულებები ყოველთვის გამოწვეულია მონაცემების შეცვლით. თუ საზიარო მონაცემების მხოლოდ ვკითხულობთ, გართულებები არ არის. თუ ერთი ან რამდენიმე დინება ცდილობს საზიარო მონაცემების შეცვლას, ჩნდება საფრთხე.

პროგრამირებაში და ალგორითმების ანალიზში ძალიან სასარგებლო არის **ინვარიანტის** ცნება. ესაა დებულება მოცემული მონაცემთა სტრუქტურის მიმართ, რაც ყოველთვის სამართლიანია. განახლების პროცესში ინვარიანტები ხშირად ირღვევა.

მაგალითად, ორმხრივ ბმული სიის ყოველი კვანძი შეიცავს პოინტერს თავის წინა და მომდევნო კვანძებზე. ერთ-ერთი ინვარიანტი იმაში მდგომარეობს, რომ თუ თქვენ მიჰყვებით „next" პოინტერს A კვანძიდან B კვანძისკენ, მაშინ B კვანძის „previous" პოინტერი მიუთითებს A კვანძს. როდესაც სიიდან კვანძის ამოღება გვინდა, ორივე მხარეზე მოთავსებული კვანძების პოინტერები უნდა გადახალისდეს (განახლდეს) ისე, რომ ერთმანეთზე მიუთითებდნენ. როდესაც მხოლოდ ერთი პოინტერი არის განახლებული, ინვარიანტი დარღვეულია ვიდრე მეორე მხარის საჭირო პოინტერიც არ განახლდება. ამის შემდეგ ინვარიანტი ისევ აღდგება. ასეთი სიიდან კვანძის წასაშლელად სამი ბიჯი არის საჭირო:

a. განვსაზღვროთ წასაშლელი კვანძის მისამართი N;
b. გადავახალისოთ ბმული N -ის წინადან N -ის მომდევნო კვანძზე;
c. გადავახალისოთ ბმული N -ის მომდევნოდან N -ის წინა კვანძზე.

b და c ბიჯებს შორის, ერთი მიმართულებით მიმავალი ბმულები არათავსებადია მეორე მიმართულებით მიმავალ ბმულებთან და ინვარიანტი დარღვეულია. თუ დარღვეული ინვარიანტის პირობებში მეორე დინებაც ცდილობს გარკვეული სამუშაოს შესრულებას იგივე კვანძებთან, მაშინ შედეგი არაცალსახაა. მაგალითად, თუ მეორე დინება ცდილობს სიის შემოვლას, შესაძლოა მან მოასწროს ყველა კვანძის შემოვლა და დამუშავება, შესაძლოა გამოტოვოს ის კვანძი, რომლის ამოღებაც მიმდინარეობს. ნებისმიერ შემთხვევაში, დარღვეული ინვარიანტი ქმნის ერთ-ერთი ყველაზე გავრცელებული შეცდომის (bug) საფუძველს ერთდროულობის (concurrent) კოდში, რასაც ეწოდება **რბოლის (ან დასწრების) ვითარება.**

<<< რბოლის ვითარება (race conditions). მასთან გამკლავება

ერთდროულ პროცესებში, რბოლის (ანუ დასწრების) ვითარება (race conditions) არის ნებისმიერი რამ, სადაც შედეგი დამოკიდებულია ორი ან მეტი დინების ოპერაციების შესრულების

მიმდევრობის რიგზე. დინებები ერთმანეთს ასწრებენ თავისი ოპერაციების შესრულებას. C++ -ის სტანდარტი აგრეთვე განსაზღვრავს ტერმინს (data race) **მონაცემების რბოლა.** ესაა რბოლის ვითარების კერძო სახე, რასაც ადგილი აქვს ერთი ობიექტის ერთდროული ცვლილებების გამო. მონაცემების რბოლა იწვევს განუსაზღვრელ ყოფაქცევას (undefined behavior).

რბოლის ვითარებებთან გამკლავების რამდენიმე გზა არის ცნობილი.

უმარტივესი არის მონაცემთა სტრუქტურის შეფუთვა დამცავი მექანიზმებით იმ მიზნით, რომ მხოლოდ ცვლილებების განმხორციელებელ დინებას ჰქონდეს იმ საშუალედო მდგომარეობების ნახვის უფლება, როდესაც ინვარიანტები დარღვეულია. სხვა დინებების თვალთახედვით, რომლებიც აღწევენ მონაცემთა სტრუქტურაში, ასეთი ცვლილებები ან არ დაწყებულა, ან უკვე დამთავრებულია.

სხვა არჩევანი არის მონაცემთა სტრუქტურის და მისი ინვარიანტების დაგეგმარების (დიზაინის) შეცვლა ისე, რომ ცვლილებები განხორციელდეს როგორც განუყოფელი ცვლილებების მიმდევრობა, რომლის თითოეული წევრი ინარჩუნებს ინვარიანტებს. ამას ეწოდება ჩაკეტვების-გარეშე პროგრამირება (lock-free programming) და ძნელია განსახორციელებლად.

კიდევ სხვა გზა არის მონაცემთა სტრუქტურის განახლების, როგორც გარიგების (transaction) დამუშავება, ისევე როგორც მონაცემთა ბაზის განახლება ხდება გარიგების დროს. ეს თემა გადის C++ ენის სტანდატრის საზღვრებიდან.

<<< საზიარო მონაცემების დაცვა მუტექსით (mutex)

მუტექსი ნიშნავს ორმხრივობის გამორიცხვას (*mut*ual *ex*clusion). იგი არის მონაცემების დაცვის ყველაზე ზოგადი მექანიზმი C++ ენაში. თუმცა მის გამოყენებას თან ახლავს თავისი საკუთარი გართულებები - ჩიხები (deadlocks) და მუტექსის რაციონალური გამოყენება ანუ ერთი მუტექსის მიერ ძალიან დიდი ან მცირე მონაცემების დაცვა.

მუტექსი არის სინქრონიზაციის პრიმიტივი (synchronization primitive), რომელიც უზრუნველყოფს ერთი დინების მიერ კოდის გარკვეული ფრაგმენტის დაკეტვას ან გაღებას სხვა დინებებისთვის. ვიდრე შევალთ საზიარო მონაცემთა სტრუქტურაში, მუტექსი იკეტება, ხოლო მუშაობის დასრულების შემდეგ მუტექსი იღება. ამის გამო სხვა დინებები ვერ ხედავენ დარღვეულ ინვარიანტებს.

> ***Synchronization primitives*** – *ესენი არიან პროგრამული უზრუნველყოფის მარტივი მექანიზმები, რასაც პლატფორმა (ოპერაციული სისტემა) სთავაზობს მომხმარებელს, დინების ან პროცესის სინქრონიზების მიზნით. ისინი იგება დაბალი დონის მექანიზმებით (ატომური ოპერაციები, მეხსიერების ბარიერები, შინაარსობრივი გადამრთველები და სხვა).*

მუტექსი არის მონაცემთა დაცვის ყველაზე ზოგადი საშუალება, მაგრამ ის არც ერთადერთია და არც იდეალური.

<<< მუტექსის გამოყენება C++ -ში

მუტექსი ჩნდება `std::mutex`-ის ობიექტის შექმნით, იკეტება `lock()`-ის გამოძახებით და იხსნება `unlock()`-ის გამოძახებით. ეს შეიძლება გაკეთდეს ცხადად, ან RAII იდიომის გამოყენებით (Resource acquisition is initialization - რესურსის დაუფლება არის ინიციალიზაცია) რაც ბევრად მოსახერხებელია, თუმცა RAII-ის განხორციელება რამდენიმენაირადაა შესაძლებელი. ვნახოთ ცხადი გამოყენების მაგალითი.

განვიხილოთ მარტივი პროგრამა, რომელშიც ორი დინება ცდილობს 50-50 სიმბოლოს დაბეჭდვას. აქ ეკრანი არის საზიარო რესურსი, რომლისთვისაც მიდის რბოლა დინებებს შორის.

```
#include <iostream>       // std::cout
#include <thread>         // std::thread

void print_block(int n, char c)
```

```
{
	// critical section
	for (int i = 0; i<n; ++i) { std::cout << c; }
	std::cout << std::endl;
}

int main()
{
	std::thread th1(print_block, 50, '*');
	std::thread th2(print_block, 50, '$');

	th1.join();
	th2.join();
}
```

სხვადასხვა გაშვებაზე სხვადასხვა რამ იბეჭდება, დაახლოებით ასეთი სახის:

```
***$$$**********************$$$$***************$$$************$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
Press any key to continue . . .
```

ახლა, გამოვიყენოთ მუტექსი ორმხრივობის გამოსარიცხად. რომელი დინებაც პირველი შევა რესურსში, ბოლომდე ისარგებლებს მისით

```
//// mutex example
#include <iostream>		// std::cout
#include <thread>		// std::thread
#include <mutex>		// std::mutex

std::mutex mtx;			// mutex for critical section

void print_block(int n, char c)
{
	// critical section (exclusive access to std::cout signaled by locking mtx):
	mtx.lock();
	for (int i = 0; i<n; ++i) { std::cout << c; }
	std::cout << std::endl;
	mtx.unlock();
}

int main()
{
	std::thread th1(print_block, 50, '*');
	std::thread th2(print_block, 50, '$');

	th1.join();
	th2.join();
}
```

ახლა, ორი ხაზი იბეჭდება, თითოზე 50 ერთი და იგივე სიმბოლო. თუმცა, ჯერ დოლარები დაიბეჭდება თუ ჯერ ვარსკვლავები - დამოკიდებულია იმაზე თუ რომელი დინება მოასწრებს კრიტიკული მონაკვეთის დაკეტვას.

ყურადღება უნდა მივაქციოთ იმ გარემოებას, რომ მუტექსი ალგორითმია. ამიტომ ორმხრივობა გამოირიცხება იმ შემთხვევაში, თუ დინებები ერთი და იმავე მუტექსს კეტავენ, ანუ ერთი და იმავე ალგორითმით მუშაობენ.

გამართლებულად არ ითვლება კოდის კრიტიკული მონაკვეთისთვის ჩაკეტვის და გახსნის ცხადად გამოძახება. მაგალითად, თუ კრიტიკულ მონაკვეთში განშტოებებია, მაშინ ყოველი შტოს გასწვრივ უნდა გამოვიძახოთ `unlock()`, გამონაკლისების ჩათვლით. გამოსავლად სტანდარტული ბიბლიოთეკა გვთავაზობს რამდენიმე თარგიან კლასს, რომლებიც

მუტექსისთვის ახორციელებენ RAII იდიომას. იგი კეტავს მიწოდებულ მუტექსს ჩამკეტი ობიექტის აგების პროცესში და აღებს მას ამ ობიექტის დაშლისას.

განვიხილოთ იგივე მაგალითი `std::lock_guard` თარგიანი კლასის გამოყენებით. საკმარისია ფუნქციის ტანში შევიტანოთ ბუნებრივი ცვლილება:

```
void print_block(int n, char c)
{
	// critical section (exclusive access to std::cout signaled by locking mtx):
	std::lock_guard <std::mutex> lck(mtx);
	for (int i = 0; i<n; ++i) { std::cout << c; }
	std::cout << std::endl;
}
```

C++17-ს შეუძლია თარგის ტიპის თვით-ამოცნობა, ამიტომ ახალ კომპილერებზე შეგვიძლია ვწეროთ უფრო მოკლე კოდი:

```
	std::lock_guard lck(mtx);
```

C++17 შეიცავს std::scoped_lock -ს, რომელიც აკეთებს იგივეს რასაც std::lock_guard, მაგრამ დამატებით შეუძლია რამდენიმე მუტექსის ერთდროული დამუშავება:

```
void print_block(int n, char c)
{
	std::scoped_lock lck(mtx);
	for (int i = 0; i<n; ++i) { std::cout << c; }
	std::cout << std::endl;
}
```

განხილულ მაგალითებში მუტექსი გლობალურ ცვლადს წარმოადგენდა. უმჯობესია ობიექტზე ორიენტირებული დიზაინის გამოყენება.

მუტექსი არის ძირითადი საშუალება სინქრონიზაციისთვის, მაგრამ მასთან ერთად საჭიროა კოდის სტრუქტურირება და მონაცემთა სტრუქტურის დიზაინის შერჩევა საზიარო მონაცემების დაცვის მიზნით.

<<< კოდის სტრუქტურირება საზიარო მონაცემების დაცვის მიზნით

მომდევნო მაგალითში ვნახავთ, თუ რამდენად სახიფათოა ფუნქციიდან დაცულ მონაცემებზე პოინტერის ან რეფერენსის დაბრუნება return-ით ან რაიმე გარე პარამეტრით. განსაკუთრებით საშიშია დაცული ფრაგმენტის შემცველი ფუნქციებიდან დაცულ მონაცემებზე პოინტერების ან რეფერენსების გადაწოდება გამოძახებული ფუნქციებისთვის. განვიხილოთ მაგალითი:

```
class some_data
{
	int a;
	std::string b;
public:
	void do_something();
};
class data_wrapper
{
private:
	some_data data;
	std::mutex m;
public:
	template<typename Function>
	void process_data(Function func)
	{
	    std::lock_guard<std::mutex> l(m);
	    func(data);            //შესაძლოა „დაცული“ მონაცემების გადაწოდება მომხმარებლის
	                           //მიერ მოწოდებული ფუნქციისთვის  (1)
	}
```

```
};
some_data* unprotected;
void malicious_function(some_data& protected_data)
{      unprotected = &protected_data;  }
data_wrapper x;
void foo()
{
    x.process_data(malicious_function); //არგუმენტად „ვერაგი“ ფუნქციის გადაწოდება      (2)
    unprotected->do_something();     //დაუცველი წვდომა დაცულ მონაცემებზე                (3)
}
```

process_data ფუნქციის კოდი უწყინარად გამოყურება, თუმცა (1) სტრიქონი გვიჩვენებს რომ მომხმარებლის მიერ მოწოდებული func ფუნქციის გამოძახების შესაძლებლობის შედეგად, foo ფუნქციას შეუძლია malicious_function ფუნქციის გადაწოდება (2) სტრიქონში დაცვის გვერდის ასავლელად და მუტექსის გვერდის ავლით do_something() ფუნქციის გამოძახება ((3)).

მსგავსი ვითარებებისთვის თავის დასაცავად უნდა ვიხელმძღვანელოთ შემდეგი წესით:

> *Don't pass pointers and references to protected data outside the scope of the lock, whether by returning them from a function, storing them in externally visible memory, or passing them as arguments to user-supplied functions - არანაირად არ გადააწოდოთ დაცული მონაცემების პოინტერები და რეფერენსები ჩაკეტილი სფეროს (ფრაგმენტის) გარეთ - არც ფუნქციიდან დაბრუნებით, არც გარედან ხილვად მეხსიერებაში შენახვით, არც მათი გადაწოდებით მომხმარებლის მიერ მოწოდებულ ფუნქციებში.*

<<< ინტერფეისებისთვის დამახასიათებელი დასწრების ვითარებების გამოკვლევა

მხოლოდ მუტექსის გამოყენება არ ნიშნავს რომ საზიარო მონაცემები დაცულია. საჭიროა დამატებითი ღონისძიებები. განვიხილოთ ბმული სიის მაგალითი. იმისათვის რომ დინებამ უსაფრთხოდ წაშალოს კვანძი, თავიდან უნდა ავიცილოთ სხვა დინებების მხრიდან ერთდროული წვდომა სამ კვანძზე: წასაშლელზე და მის ორივე მეზობელზე. მაგრამ ეს არაა საკმარისი, რბოლის ვითარება მაინც შეიძლება შეიქმნას - არა მხოლოდ კვანძებია დასაცავი, არამედ მთლიანი მონაცემთა სტრუქტურა, მთლიანი წაშლის ოპერაციისთვის. უმრატივესი გამოსავალი არის ერთი ცალი მუტექსის მოშველიება მთლიანი სიის დასაცავად.

და ახლაც, თუმცა ინდივიდუალური ოპერაციები დაცული იქნება, თქვენ მაინც შეიძლება მიიღოთ რბოლის ვითარება, ასეთი მარტივი ინტერფეისითაც კი. განვიხილოთ სტეკის მონაცემთა სტრუქტურა, std::stack კონტეინერის ადაპტერის მსგავსი ინტერფეისით:

```
template<typename T, typename Container = std::deque<T> >
class stack
{
public:
       explicit stack(const Container&);
       explicit stack(Container&& = Container());
       template <class Alloc> explicit stack(const Alloc&);
       template <class Alloc> stack(const Container&, const Alloc&);
       template <class Alloc> stack(Container&&, const Alloc&);
       template <class Alloc> stack(stack&&, const Alloc&);
       bool empty() const;
       size_t size() const;
       T& top();
       T const& top() const;
       void push(T const&);
       void push(T&&);
       void pop();
       void swap(stack&&);
       template <class... Args> void emplace(Args&&... args);
};
```

აქ რამდენიმე გართულება ჩნდება.

`empty()` და `size()`-ის შედეგები მრავალდინებიან გარემოში სანდო აღარაა. შედეგი შესაძლოა კორექტული იყოს მათი გამოძახების მომენტში, მაგრამ შედეგის დაბრუნების პროცესში, ვიდრე გამომძახებელი დინება გამოიყენებს ამ ინფორმაციას, შესაძლოა სხვა დინებებმა `push()` ახალი ელემენტები ან `pop()` არსებულები.

როდესაც `stack`-ის ნიმუში საზიარო არაა, ძალიან გავრცელებული და მისაღებია შემდეგი სახის კოდის გამოყენება:

```
stack<int> s;
   ...
if (!s.empty())                          ①
{
      int const value = s.top();         ②
      s.pop();
      do_something(value);               ③
}
```

ცარიელ სტეკზე `top()`მეთოდის გამოძახება იწვევს განუსაზღვრელ ყოფაქცევას.

საზიარო სტეკის შემთხვევაში კოდის ეს ფრაგმენტი აღარაა უსაფრთხო რამდენიმე მიზეზის გამო.

`empty()`-სა ① და `top()`-ის ② გამოძახებებს შორის შესაძლებელია სხვა დინების მიერ გამოძახებულმა `pop()`-მა ამოიღოს სტეკიდან ბოლო ელემენტი. ეს კლასიკური დასწრების ვითარებაა და სტეკის შიგნით მუტექსის გამოყენება ამ ვითარებას ვერ აგვარიდებს - ეს არის ინტერფეისის შედეგი.

გამოსავალი არის ინტერფეისის შეცვლა. მაგრამ როგორ? უმარტივეს შემთხვევაში, შეიძლებოდა `top()`-ის ისე გაკეთება, რომ მან გამოისროლოს გამონაკლისი ცარიელი სტეკის შემთხვევაში. მაგრამ ეს ართულებს კოდს. ახლა საჭირო ხდება გამონაკლისის დაჭერა იმ შემთხვევაშიც კი, როდესაც `empty()`-ის გამოძახება აბრუნებს false-ს და აგრეთვე სხვა შემთხვევების გათვალისწინება.

კოდის განხილულ ნაგლეჯში არის სხვა დასწრების ვითარების საფრთხეც. ამჯერად, `top()`-ის ② და `pop()`-ის ③ გამოძახებებს შორის. განვიხილოთ ორი დინება, რომლებიც უშვებენ კოდის ზემოთმოყვანილ ნაგლეჯს ერთი და იმავე `stack` ობიექტზე. ეს არაა უჩვეულო. როდესაც დინებებს იყენებენ წარმადობის ასამაღლებლად, გავრცელებული შემთხვევაა რომ დინებები უშვებენ ერთი და იმავე კოდს სხვადასხვა მონაცემებზე. `stack` (მაგრამ უფრო ხშირად queue) მოსახერხებელია დავალებების გასანაწილებლად დინებებს შორის.

დავუშვათ რომ თავდაპირველად სტეკში იყო ორი ელემენტი, ამიტომ შეგვიძლია ამ შევწუხდეთ `empty()`-სადა `top()`-ის გამოძახებებს შორის დასწრების გარემოს გაჩენის საკითხით.

თუ სტეკი შიგნიდან დაცულია მუტექსით, მხოლოდ ერთ დინებას შეუძლია სტეკის წევრი ფუნქციის გაშვება დროის ნებისმიერ შუალედში. ამიტომ გამოძახებები მშვენივრადაა ურთიერთშენაცვლებული. თუმცა, `do_something()`-ის გაშვება შეუძლიათ ერთდროულად. ერთი შესაძლო ვარიანტი კოდების შესრულებისა მოყვანილია შემდეგ ცხრილში:

A დინება	B დინება

`if (!s.empty())`
`  int const value = s.top();`
`  s.pop();`
`  do_something(value);`
	`if (!s.empty())`
	`  int const value = s.top();`
	`  s.pop();`
	`  do_something(value);`

თუ მხოლოდ ეს ორი დინება მუშაობს, მაშინ `top()`-ის ორ გამოძახებას შორის არაფერია რაც სტეკს შეცვლიდა. შესაბამისად, ორივე სტეკი ხედავს ერთი და იმავე მნიშვნელობას. ამათ მოყვება `pop()`-ის ზედიზედ ორი გამოძახება. ე.ი., ერთი მნიშვნელობა წაიშლება ყოველგვარი წაკითხვის გარეშე, მაშინ როდესაც მეორე ორჯერ დამუშავდება. ესაც რბოლის ვითარებაა. უფრო ვერაგი და ძნელად შესამჩნევი ვიდრე `empty()/top()`-ის რბოლა. შედეგების სერიოზულობა დამოკიდებულია იმაზე, თუ ზუსტად რას აკეთებს `do_something()` ფუნქცია.

ამგვარად, საჭიროა ინტერფეისის არსებითი ცვლილება. ერთი ცვლილება იქნება `top()`-ის და `pop()`-ის შერწყმა მუტექსის დაცვის ქვეშ. მაგრამ ამას მივყავართ ისეთი გართულებების გადაჭრის აუცილებლობამდე, რომლებისგან თავის ასარიდებლადაც სტეკის ინტერფეისში ეს ორი ოპერაცია განცალკევებული სახით წარმოადგინეს.

მდგომარეობაში გასარკვევად, განვიხილოთ `stack<vector<int>>`. ვექტორი არის დინამიკური (ცვლადი) ზომის კონტეინერი, ამიტომ მისი ასლირების დროს ბიბლიოთეკას სჭირდება საკმაო მეხსიერების დაჯავშნა გროვაში ვექტორის შიგთავსის ასლისთვის. თუ სისტემა მძიმედაა დატვირთული ან რესურსებზე არსებითი შეზღუდვებია დაწესებული, მეხსიერების გამოყოფა შეიძლება ჩავარდეს. შედეგად, ვექტორის ასლის კონსტრუქტორი გამოისვრის `std::bad_alloc` გამონაკლისს. სავარაუდოდ ეს მოხდება როდესაც ვექტორი ძალიან ბევრ ელემენტს შეიცავს. თუ დაგეგმილი იყო `pop()` ფუნქციის მიერ ამოღებული მნიშვნელობის დაბრუნება, ისევე როგორც ამ მნიშვნელობის ამოღება სტეკიდან, მაშინ იქნება პოტენციური გართულება: ამოღებული მნიშვნელობა გამომძახებელთან ბრუნდება მხოლოდ სტეკის ***ცვლილების*** შემდეგ, მაგრამ მონაცემების ასლის შექმნამ (რაც ხდება მონაცემების გამომძახებელთან დასაბრუნებლად) შესაძლოა გამოისროლოს გამონაკლისი. თუ ეს მოხდა, მაშინ მონაცემები დაკარგულია. მონაცემები ამოღებულია სტეკიდან, მაგრამ ასლის შექმნა წარუმატებელი აღმოჩნდა. ამის გამო იყო რომ სტეკის დიზაინერებმა ოპერაცია ორად გაყვეს: ზედა ელემენტის მიღება (top()) და შემდეგ მისი ამოღება სტეკიდან (pop()) - თუ უსაფრთხოდ ვერ ამოიღებთ მონაცემს, იგი რჩება სტეკში. თუ სირთულე მდგომარეობდა გროვის მეხსიერების სიმცირეში, მაშინ იქნებ აპლიკაცია გაანთავისუფლებდა ცოტაოდენ მესიერებას და სცდიდა თავიდან.

სამწუხაროდ, ასეთი გახლეჩის ფუფუნება არ გვაქვს მრავალდინებიან გარემოში. ზუსტად იგი იწვევს მონაცემების რბოლას.

ვნახოთ რა არადნები არსებობს მდგომარეობის გამოსასწორებლად.

არადანი 1: რეფერენსის გადაწოდება

უნდა გადავაწოდოთ იმ ცვლადის რეფერენსი, რომელშიც გვინდა რომ მივიღოთ ამოღებული (popped) მნიშვნელობა pop()-ის გამოძახების შემდეგ. ეს ბევრ შემთხვევაში კარგად მუშაობს, მაგრამ აქვს რამდენიმე ნაკლი. ძირითადი ისაა, რომ შესანახი მნიშვნელობა უნდა იყოს მინიჭებადი, რაც მნიშვნელოვანი შეზღუდვაა. მომხმარებლის მიერ შექმნილი ბევრი ტიპი მინიჭებადი არაა, თუმცა აქვთ გადაადგილების კონსტრუქტორი და ასლის კონსტრუქტორიც კი.

არადანი 2: NO-THROW ასლის ან გადაადგილების კონსტრუქტორის მოთხოვნა

მნიშვნელობების დამბრუნებელ pop() ფუნქციასთან არის მხოლოდ ერთი ერთი გართულება, რაც უკავშირდება გამონაკლისებს: როდესაც დასაბრუნებელი მნიშვნელობა ისვრის გამონაკლისს. ბევრ ტიპს აქვს ასლის კონსტრუქტორი რომელიც არ ისვრის გამონაკლისს და C++ -ში ახალი მარჯვენა მნიშვნელობების რეფერენსების მხარდაჭერის პირობებში კიდევ უფრო მეტ ტიპს აქვს გადაადგილების კონსტრუქტორი, რომელიც არ ისვრის გამონაკლისს - იმ შემთხვევაშიც კი თუ მათი ასლის კონსტრუქტორი ისვრის. ერთი გამართული არადანი იქნება თუ შევზღუდავთ ჩვენი დინებებიგან დაცულ (thread-safe) სტეკს იმ ტიპებით, რომლებიც უსაფრთხოდ აბრუნებს მნიშვნელობას გამონაკლისის გამოსროლის გარეშე.

თუმცა ეს უსაფრთხოა, მაგრამ იდეალური არაა. ეს მუშაობს, რადგან კომპილაციის დროს შეგვიძლია გავარკვიოთ ასლის ან გადაადგილების კონსტრუქტორი ისვრის თუ არა გამონაკლისს (`std::is_nothrow_copy_constructible` და `std::is_nothrow_move_constructible`-ების გამოყენებით). მაგრამ მომხმარებლის მიერ განსაზღვრულ ტიპებში ბევრად მეტია ისეთები, რომლებსაც აქვთ ასლის კონსტრუქტორი გამონაკლისის გამოსროლით და არ აქვთ გადაადგილების კონსტრუქტორი (თუმცა, დროთა განმავლობაში ეს რაოდენობა იცვლება იმისდა მიხედვით, თუ როგორ ეჩვევა ადამიანები C++11 – ის მარჯვენა მნიშვნელობების რეფერენსების მხარდაჭერას). არ იქნება კარგი, თუ ასეთ ტიპებს ვერ შევინახავთ ჩვენს სტეკში.

არადანი 3: დავაბრუნოთ ამოღებული მნიშვნელობის პოინტერი

აქ პლიუსი ისაა, რომ პოინტერის ასლი კეთდება გამონაკლისის გამოსროლის გარეშე. მინუსი ისაა, რომ პოინტერს უკავშირდება მეხსიერების გამოყოფის საკითხები და ისეთი მარტივი ტიპებისთვის, როგორიცაა მთელები, მეხსიერების გამოყოფის ზედნადები ხარჯები შესაძლოა აჭარბებდეს მნიშვნელობის დაბრუნების ხარჯს. თუ ინტერფეისი იყენებს ამ არადანს, მისთვის `std::shared_ptr` იქნება პოინტერის სრულიად მისაღები სახე. იგი არ იწვევს მეხსიერების გაჟონვას არაა საჭირო new, delete შეტყობინებების გამოყენება. ეს მნიშვნელოვანია, რაგან new, delete-ის გამოყენებები საგრძნობ ზედნადებს იწვევს საწყის დინებებისგან დაუცველ ვერსიასთან შედარებით.

არადანი 4: ერთდროულად არადანი 1-ის და არადანი 2 ან 3-ის გათვალისწინება

თუ აირჩიეთ არადანი 2 ან 3, მაშინ არადანი 1-ის დამატება შედარებით იოლია და ეს აძლევს თქვენი კოდის მომხმარებლებს საშუალებას რომ აირჩიონ თუ რომელი არადანი უფრო მისაღებია მათთვის მცირე დამატებითი ხარჯის ფასად.

დინებებისგან დაცული სტეკის განსაზღვრის მაგალითი

ლისტინგი 1 გვიჩვენებს სტეკისთვის განსაზღვრულ კლასს, რომელსაც არ შეექმნება რბოლის გარემო და რომელიც განახორციელებს 1 და 3 არადანებს. აქ არის ორჯერ გადატვირთული pop(), ერთი მიიღებს იმ ადგილის რეფერენსს სადაც უნდა შევინახოთ ამოღებული მნიშვნელობა და მეორე რომელიც აბრუნებს `std::shared_ptr<>`-ს. სტეკის ინტერფეისში ფუნქციების რაოდენობა შემცირებულია.

ლისტინგი 2.1. დინებებისგან დაცული სტეკის კლასის მონახაზი

```
#include <exception>
#include <memory>
struct empty_stack : std::exception
{
	const char* what() const noexcept;
};
template<typename T>
class threadsafe_stack
{
public:
	threadsafe_stack();
	threadsafe_stack(const threadsafe_stack&);
	threadsafe_stack& operator=(const threadsafe_stack&) = delete;
	void push(T new_value);
	std::shared_ptr<T> pop();
	void pop(T& value);
	bool empty() const;
}
```

ინტერფეისის შეკვეცით ვაღწევთ მაქსიმალურ უსაფრთხოებას. სტეკის მინიჭება არ შეიძლება, რადგან მინიჭება გაუქმებულია. არ გვაქვს მნიშვნელობების გადაცვლის ფუნქცია (swap). სტეკის ასლის გაკეთება შეიძლება და ეს მისაღებია როდესაც სტეკში ასლირებადი ელემენტებია. pop() ფუნქცია ისვრის გამონაკლისს როდესაც სტეკი ცარიელია. ამიტომ ყველაფერი მუშაობს მაშინაც,

როდესაც სტეკი იცვლება empty()-ის გამოძახების შემდეგ. სტეკის ხუთი ოპერაციიდან დარჩა 3 და სინამდვილეში empty() ზედმეტიცაა. ინტერფეისის გამარტივება აუმჯობესებს მონაცემების მითვალყურების ხარისხს. შემდეგი ლისტინგი გვაძლევს სტეკის შემფუთველის (wrapper) იმპლემენტაციას.

ლისტინგი 2.2. ხორცშესხმული დინებებისგან დაცული სტეკი

```
#include <exception>
#include <memory>
#include <mutex>
#include <stack>
struct empty_stack : std::exception
{
	const char* what() const throw()
	{
		return "Exception: Stack is empty!";
	}
};
template<typename T>
class threadsafe_stack
{
private:
	std::stack<T> data;
	mutable std::mutex m;
public:
	threadsafe_stack() {}
	threadsafe_stack(const threadsafe_stack& other)
	{
		std::lock_guard<std::mutex> lock(other.m);
		data = other.data;
	}
	threadsafe_stack& operator=(const threadsafe_stack&) = delete;
	void push(T new_value)
	{
		std::lock_guard<std::mutex> lock(m);
		data.push(new_value);
	}
	std::shared_ptr<T> pop()
	{
		std::lock_guard<std::mutex> lock(m);
		if (data.empty()) throw empty_stack(); //ვიდრე ამოვიღებთ, ჯერ ვამოწმებთ
		std::shared_ptr<T> const res(std::make_shared<T>(data.top()));
		//ზედა სტრიქონში, სტეკის შეცვლამდე ჯერ დასაბრუნებელ მნი-ბას განვათავსებთ
		data.pop();
		return res;
	}
	void pop(T& value)
	{
		std::lock_guard<std::mutex> lock(m);
		if (data.empty()) throw empty_stack();
		value = data.top();
		data.pop();
	}
	bool empty() const
	{
		std::lock_guard<std::mutex> lock(m);
		return data.empty();
	}
};
```

როდესაც ერთი მუტექსი იცავს საჭიროზე მეტ მონაცემს, ამბობენ რომ გვაქვს მსხვილად დაფქული ჩაკეტვის სქემა (coarse-grained locking scheme). როდესაც საჭიროს, ან ნაკლებს იცავს, ამბობენ რომ გვაქვს წვრილად-დაფქული ჩაკეტვის სქემა (fine-grained locking scheme). მაგალითად, ერთზე მეტი მუტექსის გამოყენება საჭიროა, როდესაც თითო მუტექსი კლასის თითო ობიექტს იცავს. თუ დაგვჭირდება ორი ან მეტი მუტექსის გამოყენება მოცემული ოპერაციისთვის, მაშინ ჩნდება ახალი საფრთხე - ჩიხი (deadlock). ეს არის რბოლის ვითარების საპირისპირო რამ. იმის მაგივრად რომ ორი დინება ეჯიბრებოდეს ერთმანეთს თუ რომელი იქნება პირველი, თითოეული მათგანი უცდის მეორეს და ვერცერთი ვერ აკეთებს საქმეს.

<< ჩიხი: სირთულე და გამოსავალი

წარმოვიდგინოთ, რომ წყვილი დინებიდან თითოეული ცდილობს დაკეტოს წყვილი მუტექსიდან ორივე, თითოეულს ჩაკეტილი აქვს ერთი მუტექსი და ელოდება მეორეს. ვერცერთი დინება ვერ მუშაობს, რადგან თითოეული ელოდება რომ მეორემ გაათავისუფლოს მისი მუტექსი. ასეთ სცენარს ეწოდება **ჩიხი** (deadlock), და არის ყველაზე დიდი გართულება, როდესაც გვინდა ორი ან მეტი მუტექსის ჩაკეტვა რაიმე მოქმედების შესრულების მიზნით.

ჩიხის თავიდან ასარიდებლად ზოგადი რჩევა არის რომ მუტექსები ყოველთვის ჩავკეტოთ ერთი და იმავე მიმდევრობით. თუ (ორი მუტექსის შემთხვევაში) თქვენ A მუტექსს ყოველთვის კეტავთ B მუტექსის წინ, მაშინ არასოდეს მიიღებთ ჩიხს.

შემთხვევა, როდესაც კლასის სხვადასხვა ობიექტებს სხვადასხვა მუტექსი იცავს, შედარებით რთულად ითვლება. C++ -ის სტანდარტულ ბიბლიოთეკას აქვს რამდენიმე საშუალება ამ შემთხვევისთვის. ვნახოთ თუ როგორ მოვიქცეთ შემფუთავი კლასის ობიექტებზე swap-ის განხორციელებისთვის. ვთქვათ გვაქვს კლასი და მისი შეფუთვა, რომელიც თითო მუტექსით იცავს კლასის თითო ობიექტს:

```
class some_big_object;
void swap(some_big_object& lhs, some_big_object& rhs);
class X
{
private:
	some_big_object some_detail;
	std::mutex m;
public:
	X(some_big_object const& sd) :some_detail(sd) {}
	friend void swap(X& lhs, X& rhs)
	{
		if (&lhs == &rhs)
			return;
		std::scoped_lock guard(lhs.m, rhs.m);
		swap(lhs.some_detail, rhs.some_detail);
	}
};
```

ყველაზე ბუნებრივი არის `scoped_lock`-ის გამოყენება. იგი კეტავს ორ მუტექსს და აქვს მარტივი სინტაქსი რადგან იყენებს ტიპის თვით-ამოცნობას.

უფრო ძველ კომპილერზე მოგვიწევს, მაგალითად, ასეთი კოდის გამოყენება:

```
friend void swap(X& lhs, X& rhs)
{
	if (&lhs == &rhs)
		return;
	std::lock(lhs.m, rhs.m);                                    ❶
	std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);  ❷
	std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);  ❸
	swap(lhs.some_detail, rhs.some_detail);
```

```
}
```

თავიდან, არგუმენტები მოწმდება - რომ მართლაც განსხვავებული ნიმუშებია - ისეთი მუტექსის ჩაკეტვა, რომელიც უკვე ჩაკეტილია, იწვევს განუსაზღვრელ ყოფაქცევას. ასეთი მუტექსებიც არსებობს, ესაა `std::recursive_mutex`, მაგრამ ისინი განსხვავებული მიზნით გამოიყენება.

შემოწმების შემდეგ, `std::lock`-ის ❶ გამოძახება ჩაკეტავს ორ მუტექსს და იქმნება `std::lock_guard` -ის ❷ და ❸ ნიმუშები, თითო თითო მუტექსზე. `std::adopt_lock` პარამეტრი დამატებით მიეწოდება მუტექსს და უჩვენებს `std::lock_guard` ობიექტს რომ მუტექსები უკვე დაკეტილია და მათ უნდა მიიღონ საკუთრების უფლება არსებულ მუტექსზე იმის ნაცვლად რომ კონსტრუქტორში სცადონ მუტექსის ჩაკეტვა.

სხვა შემთხვევებში, როდესაც განხილული მიდგომები არ შველის, უნდა ვისარგებლოთ რამდენიმე შედარებით მარტივი წესით რაც დაგვეხმარება ჩიხისგან თავისუფალი კოდის შექმნაში.

<<< გზამკვლევები ჩიხის ასარიდებლად

ყველაზე ხშირად ჩიხს იწვევს ჩაკეტვები. ჩიხის მიღება შეგვიძლია ორი დინებით და არავითარი ჩაკეტვით, თუ ყოველი დინება გამოიძახებს `join()`-ს მეორისთვის. ასეთი რამეები ხდება მეტი დინების შემთხვევაშიც, სამი ან მეტი დინების ციკლებმაც შეიძლება შექმნან ჩიხები. ჩიხის ასარიდებელი ყველა გზამკვლევი რჩევა დადის ერთ იდეაზე:

ნუ დაელოდებით სხვა დინებას, თუ არის შანსი რომ იგი გელოდებათ თქვენ.

კერძო გზამკვლევები გვიჩვენებენ გზებს, თუ როგორ მოვახერხოთ რომ სხვა დინება არ დაგელოდოთ თქვენ.

აერიდეთ ჩალაგებულ ჩაკეტვებს

პირველი იდეა უმარტივესია. ნუ მოითხოვთ ჩაკეტვას თუ უკვე გიკავიათ ერთი. თუ თქვენ მაინც გინდათ ჯერადი ჩაკეტვები, გააკეთეთ ეს ერთჯერადი მოქმედებით `std::lock`-ის ან სხვა საშუალების გამოყენებით.

აერიდეთ მომხმარებლის მიერ მოწოდებული კოდის გამოძახებას ვიდრე გიჭერიათ ჩაკეტვა

ეს არის წინა გზამკვლევის მარტივი შედეგი. რა იცით რა იქნება მომხმარებლის მიერ მოწოდებულ კოდში, რადგან თუ იქ მოითხოვენ ჩაკეტვას, გამოვა პირველი წესის უგულებელყოფა შესაბამისი შედეგებით.

ზოგჯერ ამის გაკეთება აუცილებელია. მაგალითად, თუ თქვენ წერთ განზოგადებულ (generic) კოდს, ისეთს როგორიცაა სტეკი ლისტინგ 2.2-ში, პარამეტრის ტიპში (ან ტიპებში) ყოველი ოპერაცია არის მომხმარებლის მიერ მოწოდებული. ამ შემთხვევაში საჭირო ხდება ახალი გზამკვლევი.

ჩაკეტვები მოითხოვეთ ფიქსირებული რიგით

თუ სრულიად აუცილებელია ორი ან მეტი ჩაკეტვის მოთხოვნა, მაგრამ არ შეიძლება ამის ერთ მოქმედებაში გაკეთება, მაშინ საუკეთესო გამოსავალი არის ყოველი დინების მიერ ჩაკეტვების ყოველთვის ერთი და იმავე რიგით მოთხოვნა.

ზოგჯერ ამის გაკეთება მარტივია. მაგალითად, განხილული სტეკისთვის მუტექსი შინაგანია ყოველი ნიმუშისთვის, მაგრამ მონაცემებში მოთავსებული მოქმედებები შეიცავს მომხმარებლის მიერ დაწერილ კოდს. ამ დროს შეგიძლიათ დაადოთ შეზღუდვა, რომ სტეკში მოთავსებული მონაცემებიდან არცერთი მოქმედება არ უნდა განხორციელდეს უშუალოდ სტეკზე. ფორმალურად ეს შეზღუდვაა, მაგრამ არც არის ბუნებრივი რომ კონტეინერში მოთავსებულმა მონაცემებმა სცადონ კონტეინერში შეღწევა.

სხვა შემთხვევებში ეს უფრო რთულია. მაგალითად, ბმული სიის დაცვისთვის ერთი გზა არის რომ თითო კვანძზე თითო მუტექსი ვიქონიოთ. ახლა, იმისათვის რომ შეაღწიონ სიაში, დინებები

მოითხოვენ ყოველი კვანძის ჩაკეტვას, რომლითაც არიან დაინტერესებულები. იმისათვის რომ დინებამ წაშალოს კვანძი, მან უნდა დაკეტოს სამი კვანძი - ამოსაღები და მისი ორივე მეზობელი. მსგავსად, სიის შემოვლისას დინებამ უნდა დაიჭიროს ჩაკეტვა მოცემულ კვანძე სანამ მიაღწევს მომდევნო კვანძის ჩაკეტვას, რათა დარწმუნდეს რომ მომდევნო პოინტერი არაა შეცვლილი ამასობაში. მას მერე რაც მომდევნო კვანძის ჩაკეტვა გამხორციელდება, მის წინა კვანძე ჩაკეტვა მოიხსნება.

როდესაც რამდენიმე დინება ასეთი ხელი-ხელზე ჩაკეტვის სტილით მუშაობს, ჩიხის ასაცილებლად ყველა მათგანი უნდა მოძრაობდეს (კეტავდეს კვანძებს) ერთი და იგივე, ფიქსირებული მიმართულებით. თუ ორი დინება ცდილობს სიის შემოვლას საპირისპირო მიმარტულებებით და ასეთი ხელი-ხელზე ჩაკეტვებით, ისინი წააწყდებიან ჩიხს სადღაც შუაში. ვთქვათ A და B არიან მეზობელი კვანძები და ერთი მიმართულებით მოძრავი დინება იჭერს ჩაკეტვას A-ზე და ცდილობს B-ს დაკეტვას მაშინ როდესაც საპირისპირო მიმართულებით მოძრავი დინება იჭერს ჩაკეტილ B-ს და ცდილობს A-ს დაკეტვას - კლასიკური სცენარია ჩიხისთვის.

ჩაკეტვების იერარქიის გამოყენება

ეს არის კერძო შემთხვევა და დეტალებში არ შევალთ. ასეთი იერარქია საშუალებას იძლევა რომ პროგრამის მსვლელობისას თვალყური ვადევნოთ ჩაკეტვების მიმდევრობაზე გარკვეული შეთანხმების დაცულობას. იდეა მდგომარეობს აპლიკაციის გარკვეულ შრეებად დაყოფაში. მუტექსებიც ამოიცნობა თუ რომელ შრეზე უნდა იმუშაოს. როდესაც კოდი ცდილობს მუტექსის ჩაკეტვას, იგი ამას ახერხებს მხოლოდ იმ შემთხვევაში თუ მუტექსი არაა დაკეტილი ან დაკეტილია ქვედა დონის ჩაკეტვით. C++ ენის სტანდარტული ბიბლიოთეკა არ გვთავაზობს პირდაპირ მხარდაჭერას ასეთი შემთხვევებისთვის, მაგრამ მომხმარებლის მიერ შექმნილი `hierarchical_mutex`-ების კოდები არსებობს, მათ შორის ჩვენს ძირითად წყაროში, და მათი გამოყენება შესაძლებელია.

ზემოთ მოყვანილი გზამკვლევების გავრცელება ჩაკეტვებიდან სხვა სცენარებზე

ჩიხები მრავალი მიზეზით იქმნება. ამიტომ მნიშვნელოვანია ამ დროსაც გვქონდეს გზამკვლევი სახელმძღვანელო მითითებები. მომდევნო პუნქტებში განვიხილავთ რამდენიმე შემთხვევას.

<<< მოქნილი ჩაკეტვა std::unique_lock -ით

std::unique_lock შედარებით მოქნილია ვიდრე `std::lock_guard`, რადგან მას შეუძლია ინვარიანტების შემსუბუქება - std::unique_lock ყოველთვის არაა იმ მუტექსის მესაკუთრე, რომელთანაც არის დაკავშირებული. ჩვენ ვნახეთ რა იყო `std::adopt_lock`, რომლის გდაწოდება std::unique_lock -ისთვის შეიძლება მეორე არგუმენტად. მეორე არგუმენტად შეიძლება აგრეთვე `td::defer_lock` -ის გადაწოდებაც. შემდეგი კოდი იგივე საქმეს აკეთებს რაც ადრე ვნახეთ სხვა საშუალებების გამოყენებით. თუმცა მოქნილობის საფასურად, std::unique_lock ოდნავ ნელია.

```
class some_big_object;
void swap(some_big_object& lhs, some_big_object& rhs);
class X
{
private:
	some_big_object some_detail;
	std::mutex m;
public:
	X(some_big_object const& sd) :some_detail(sd) {}
	friend void swap(X& lhs, X& rhs)
	{
		if (&lhs == &rhs)
			return;
		std::unique_lock<std::mutex> lock_a(lhs.m, std::defer_lock);
		std::unique_lock<std::mutex> lock_b(rhs.m, std::defer_lock);
		std::lock(lock_a, lock_b);
```

```
            swap(lhs.some_detail, rhs.some_detail);
        }
};
```

`std::unique_lock<std::mutex> lock_a(lhs.m, std::defer_lock);` და მის მომდევნო სტრიქონში, `std::defer_lock` მუტექსს არ კეტავს, ტოვებს ღიას. თვითონ `std::unique_lock` ობიექტების გადაწოდება შესაძლებელია `std::lock()` შემფუთავისთვის, რადგან `std::unique_lock` -ს აქვს მეთოდები `lock()`, `try_lock()`, `unlock()`. ისევე როგორ ადრე განხილულ მსგავს ამოცანაში, ამ შემთხვევაშიც უფრო ხელსაყრელია `std::scoped_lock` -ის გამოყენება თუ კომპილერი თანამედროვეა.

შევნიშნოთ აგრეთვე, რომ `std::unique_lock` არის გადაადგილებადი და არა ასლირებადი ტიპი. ამასთან, აუცილებელი არაა რომ ჩაკეტვები მათთან დაკავშირებული მუტექსების მესაკუთრეები იყვნენ. ეს საშუალებას გვაძლევს რომ მუტექსის საკუთრების უფლება გადაცემულ იქნას `std::unique_lock`-ის სხვადასხვა ნიმუშებს შორის გადაადგილების საშუალებით.

<<< ჩაკეტვის სამოქმედო სივრცე

წვრილად დაქუცმაცებული ჩაკეტვა მცირე მოცულობის მონაცემებს იცავს, ხოლო მსხვილად დაქუცმაცებული ჩაკეტვა- დიდი მოცულობის მონაცემებს. მნიშვნელოვანია განვსაზღვროთ ჩაკეტვის სიმსხვილე, და ასევე აუცილებელია ზუსტად განვსაზღვროთ ის ოპერაციები, რომლებსაც აუცილებლად სჭირდება ჩაკეტვა.

ვთქვათ რამდენიმე დინება ელოდება რაიმე საზიარო რესურსს. თუ თითოეული დინება აუცილებელზე მეტ ხანს აჩერებს ჩაკეტვას, ეს გაზრდის მოცდის მთლიან დროს. როდესაც შესაძლებელია, მუტექსი ჩაკეტეთ მხოლოდ საზიარო მონაცემებზე შეღწევის შემდეგ, ხოლო მონაცემების დამუშავება სცადეთ ჩაკეტვის გარეთ. კერძოდ, ნუ განახორციელებთ დროის შთანმთქმელ მოქმედებებს (ფაილების I/O და სხვა) ჩაკეტილ კოდში. ფაილების I/O ასჯერ და მეტად ნელია ვიდრე იგივე მოცულობის მონაცემების წაკითვა მეხსიერებიდან.

მაგალითად, `std::unique_lock` -ისთვის შეგიძლიათ გამოიძახოთ `unlock()` როდესაც კოდს აღარ სჭირდება საზიარო მონაცემებში შეღწევა და შემდეგ ისევ გამოიძახოთ `lock()` თუ შეღწევა ისევ გახდა საჭირო. მაგალითად:

```
void get_and_process_data()
{
        std::unique_lock<std::mutex> my_lock(the_mutex);
        some_class data_to_process = get_next_data_chunk();
        my_lock.unlock();
        result_type result = process(data_to_process);
        my_lock.lock();
        write_result(data_to_process, result);
}
```

`result_type result = process(data_to_process);` სტრიქონს არ სჭირდება მუტექსის ჩაკეტვა, რადგან სავარაუდოდ ეს ხანგრძლივი მოქმედებაა არასაზიარო მონაცემებზე. ამიტომ წინა სტრიქონში ხელით ვხსნით მუტექსს, ხოლო მომდევნო სტრიქონში ისევ ხელით ვკეტავთ. ეს მაგალითი აგებულია ზოგადი სახელმძღვანელო წესის შესაბამისად:

> *a lock should be held for only the minimum possible time needed to perform the required operations- ჩაკეტვა უნდა გაგრძელდეს შეძლებისდაგვარად მინიმალურ დროით, რაც აუცილებელი იქნება საჭირო მოქმედებების შესასრულებლად*

<<< საზიარო მონაცემების დაცვა კერძო შემთხვევებში: ინციალიზაციის პროცესის დაცვა

წარმოვიდგინოთ, რომ რამდენიმე დინება მუშაობს საზიარო რესურსთან. ცხადია, საკმარისია რომ მხოლოდ ერთმა მათგანმა მოახდინოს მისი ინციალიზება; ან რაიმე მსგავსი შემთხვევები რაც

განხილულია მომდევნო მაგალითებში. ასეთი შემთხვევებისთის სტანდარტულ ბიბლიოთეკას აქვს `std::once_flag` და `std::call_once`. განვიხილოთ მათი გამოყენების მარტივი მაგალითები https://en.cppreference.com/w/cpp/thread/call_once-ის მიხედვით.

`std::call_once` შეასრულებს გამოძახებდ ობიექტს მხოლოდ ერთხელ, მაშინაც კი თუ რამდენიმე დინება ერთდროულად იძახებს. უფრო დაწვრილებით:

- თუ `std::call_once` ერთხელ უკვე არის გამოძახებული, ალამი (flag) აჩვენებს ამ ფაქტს და `std::call_once` მაშინვე ბრუნდება. ასეთ გამოძახებას ეწოდება პასიური.
- სხვა შემთხვევაში გამოძახებას ეწოდება აქტიური. თუ გამოძახება გამოისვრის გამონაკლისს, იგი გავრცელდება `std::call_once` -ის გამომძახებლამდე და ალამი არ გადაბრუნდება, ასე რომ სხვა გამოძახებები შესრულდება (ასეთ გამოძახებას ეწოდება გამონაკლისის სახის). თუ გამოძახება ნორმალურად დაბრუნდება, მაშინ ალამი გადაბრუნდება და ყველა შემდეგი გამოძახება გარანტირებულად იქნება პასიური.

ვნახოთ მაგალითი:

```
#include <iostream>
#include <thread>
#include <mutex>
std::once_flag flag1, flag2;
void simple_do_once()
{
    std::call_once(flag1, []() { std::cout << "Simple example: called once\n"; });
}
void may_throw_function(bool do_throw)
{
    if (do_throw) {
       std::cout << "throw: call_once will retry\n"; // this may appear more than once
       throw std::exception();
    }
    std::cout << "Didn't throw, call_once will not attempt again\n"; // guaranteed once
}
void do_once(bool do_throw)
{
    try {
       std::call_once(flag2, may_throw_function, do_throw);
    }
    catch (...) {
    }
}

int main()
{
       std::thread st1(simple_do_once);
       std::thread st2(simple_do_once);
       std::thread st3(simple_do_once);
       std::thread st4(simple_do_once);
       st1.join();
       st2.join();
       st3.join();
       st4.join();

       std::thread t1(do_once, true);
       std::thread t2(do_once, true);
       std::thread t3(do_once, false);
       std::thread t4(do_once, true);
       t1.join();
       t2.join();
       t3.join();
```

```
    t4.join();
}
```

შედეგით

```
Simple example: called once
throw: call_once will retry
throw: call_once will retry
Didn't throw, call_once will not attempt again
```

ან

```
Simple example: called once
throw: call_once will retry
Didn't throw, call_once will not attempt again
```

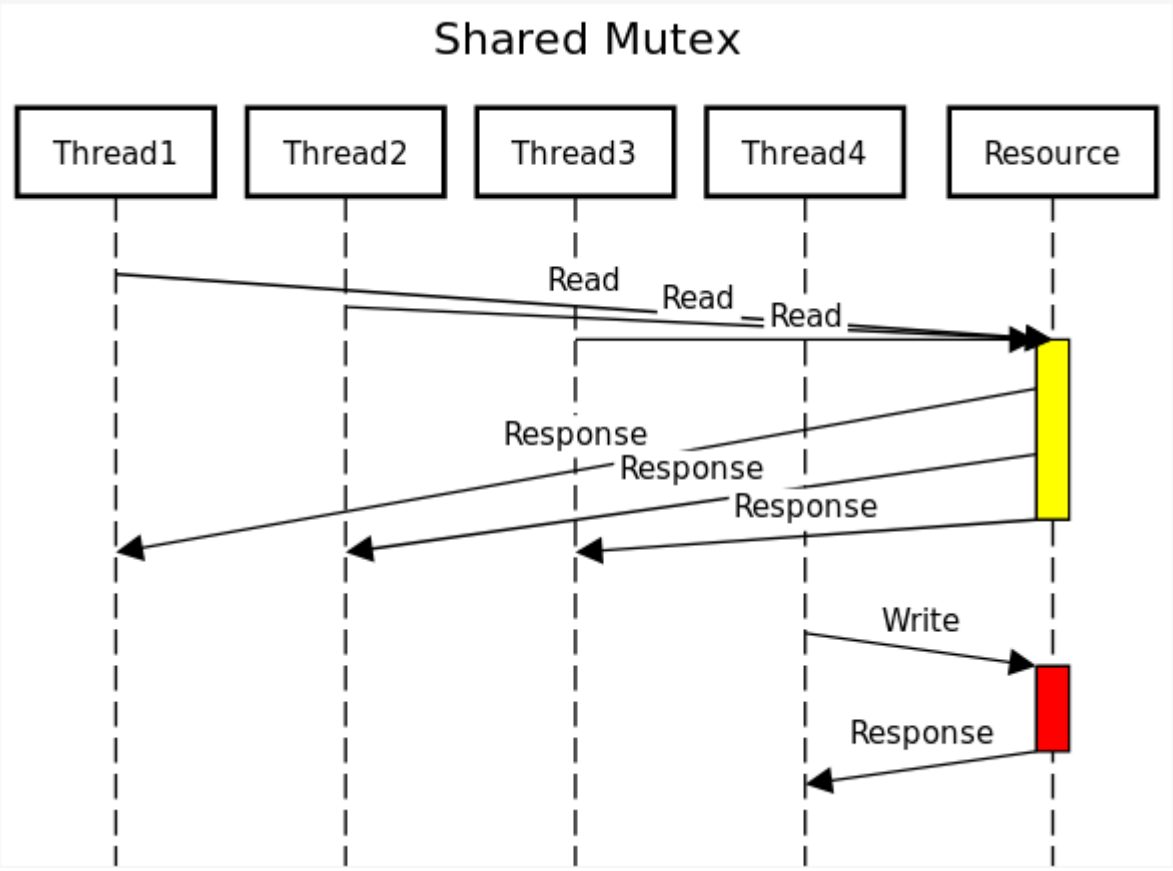
იმისდა მიხედვით თუ რომელი დინება მოასწრებს შესრულებას.

<<< საზიარო მონაცემების დაცვა კერძო შემთხვევებში: იშვიათად განახლებადი მონაცემების დაცვა

საკითხi გავაშუქოთ https://ncona.com/2019/03/read-write-mutex-with-shared_mutex/-ის მიხედვით.

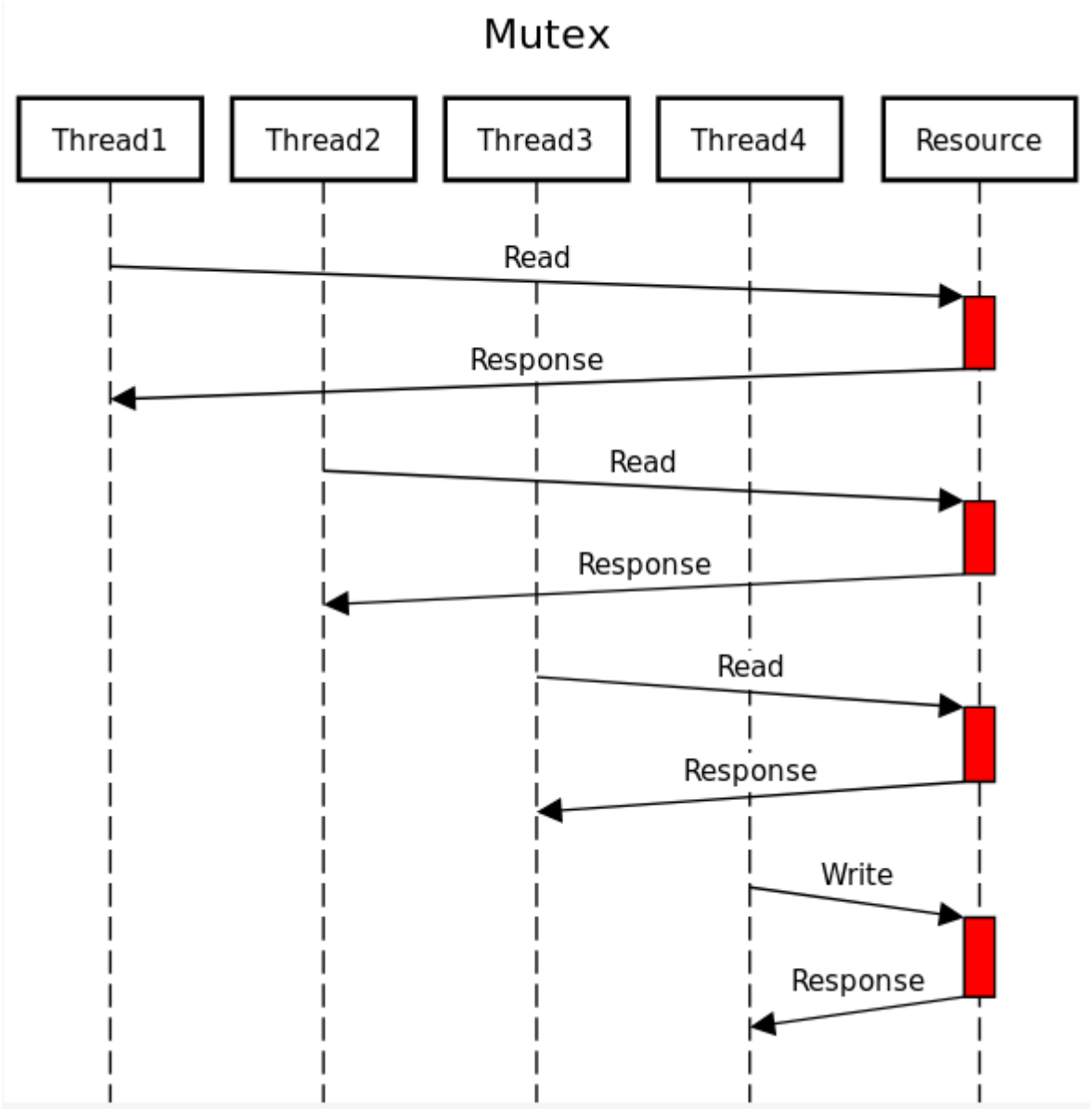
აქამდე განხილული მუტექსები მხოლოდ ერთ დინებას აძლევდა საზიარო მონაცემებზე წვდომას. ამავე დროს, გვაქვს მუტექსის ისეთი სახეობა, რომელიც აუმჯობესებს წარმადობას როდესაც საზიარო მონაცემებიდან წაკითხვები უფრო ხშირია ვიდრე ჩაწერები. ამას უწოდებენ კითხვა-წერის მუტექსს (read-write mutex), თუმცა C++ ენაში მას ჰქვია საზიარო მუტექსი.

შევხედოთ და ვნახოთ თუ რატომ არის საზიარო მუტექსი უმჯობესი მრავალი წამკითხველის პირობებში:



სურათი გვიჩვენებს, რომ მრავალ წამკითხველს შეუძლია ერთდროულად ესტუმროს საზიარო რესურსს. მაგრამ, როდესაც ჩამწერი იჭერს რესურსს, ვეღარავინ შედის იქ. თუ გამოვიყენებდით

ჩვეულებრივ მუტექსს, პროგრამა უფრო ნელა იმუშავებდა (სურათის სიმაღლე აღწერს შესრულების დროს), რადგან ყველა დინებას მოუწევდა დალოდება მუტექსისთვის:



C++17 -მა შემოიტანა საზიარო მუტექსი (shared_mutex) . ჩვეულებრივ მუტექსს აქვს სამი მეთოდი: **`lock, unlock, try_lock.`** საზიარო მუტექსი უმატებს კიდევ სამს:

`lock_shared`, **`unlock_shared`**, **`try_lock_shared.`**

პირველი სამი მუშაობს ზუსტად ისე როგორც ჩვეულებრივი მუტესის შემთხვევაში. საინტერესოა, რომ თუ მხოლოდ წამკითხველი დინებებია, მაშინ არცერთს არ სჭირდება მოცდა.

მუტექსებს შორის განსხვავება ცხადად რომ გამოჩნდეს, ჯერ განვიხილოთ სცენარი, როდესაც უმეტესობა წამკითხავია და მხოლოდ ერთია ჩამწერი. მაგრამ ჩიხის საკითხი მოვაგვაროთ ჩვეულებრივი მუტექსით:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

int value = 0;
std::mutex mutex;
```

```
// Reads the value and sets v to that value
void readValue(int& v) {
	mutex.lock();
	// Simulate some latency
	std::this_thread::sleep_for(std::chrono::seconds(1));
	v = value;
	mutex.unlock();
}

// Sets value to v
void setValue(int v) {
	mutex.lock();
	// Simulate some latency
	std::this_thread::sleep_for(std::chrono::seconds(1));
	value = v;
	mutex.unlock();
}

int main()
{
	int read1;
	int read2;
	int read3;

	auto st = std::chrono::high_resolution_clock::now();

	std::thread t1(readValue, std::ref(read1));
	std::thread t2(readValue, std::ref(read2));
	std::thread t3(readValue, std::ref(read3));
	std::thread t4(setValue, 1);

	t1.join();
	t2.join();
	t3.join();
	t4.join();

	auto diff = std::chrono::high_resolution_clock::now() - st;
	auto time = std::chrono::duration_cast<std::chrono::milliseconds>(diff);
	std::cout << std::endl << "time: " << time.count() << std::endl;

	std::cout << read1 << "\n";
	std::cout << read2 << "\n";
	std::cout << read3 << "\n";
	std::cout << value << "\n";
}
```

გავზომოთ დრო. დაახლოებით 4 წამია, რადგან ყოველ დინებას სჭირდება ერთი წამი.

ახლა გადავახალისოთ პროგრამა shared_mutex-ის გამოყენებით:

```
#include <iostream>
#include <thread>
#include <shared_mutex>
#include <chrono>

int value = 0;
std::shared_mutex mutex;

// Reads the value and sets v to that value
void readValue(int& v)
{
	mutex.lock_shared();
```

```
	// Simulate some latency
	std::this_thread::sleep_for(std::chrono::seconds(1));
	v = value;
	mutex.unlock_shared();
}

// Sets value to v
void setValue(int v)
{
	mutex.lock();
	// Simulate some latency
	std::this_thread::sleep_for(std::chrono::seconds(1));
	value = v;
	mutex.unlock();
}
int main()
{
	int read1;
	int read2;
	int read3;

	auto st = std::chrono::high_resolution_clock::now();

	std::thread t1(readValue, std::ref(read1));
	std::thread t2(readValue, std::ref(read2));
	std::thread t3(readValue, std::ref(read3));
	std::thread t4(setValue, 1);

	t1.join();
	t2.join();
	t3.join();
	t4.join();

	auto diff = std::chrono::high_resolution_clock::now() - st;
	auto time = std::chrono::duration_cast<std::chrono::milliseconds>(diff);
	std::cout << std::endl << "time: " << time.count() << std::endl;

	std::cout << read1 << "\n";
	std::cout << read2 << "\n";
	std::cout << read3 << "\n";
	std::cout << value << "\n";
}
```

ეს პროგრამა უკვე 2 წამში სრულდება. პირველი სამი დინება ერთდროულად მუშაობს და 1 წამს იკავებენ. მხოლოდ მეოთხე დინება იცდის 1 წამს.

საზიარო მუტექსისთვისაც არსებობს შემფუთავი კლასი, **`std::lock_guard`**-ის მსგავსი, რომელსაც ჰქვია **`shared_lock`**.

<<< სავარჯიშოები