ჩაკეტვაზე დაფუძნებული მონაცემთა სტრუქტურები

განხილული საკითხები:

- რას ნიშნავს ერთდროულობისთვის სტრუქტურის აგება?
- სახელმძღვანელო ერთდროულობისთვის მონაცემთა სტრუქტურის დასაგეგმარებლად >>>
- ჩაკეტვაზე დაფუძნებული მონაცემთა სტრუქტურები >>>
- დინება-უსაფრთხო სტეკი ჩაკეტვებით >>>
- დინება-უსაფრთხო რიგი ჩაკეტვებით და პირობიანი ცვლადებით >>>
- დინება-უსაფრთხო რიგი წვრილად დაქუცმაცებული ჩაკეტვებით და პირობიანი ცვლადებით >>>
- ერთდროულობის დაშვება მონაცემების განცალკევების გზით >>>
- სიიდან წევრის ამოღებაზე დალოდება >>>

სავარჯიშოები >>>

დაპროგრამებაში, მონაცემთა სტრუქტურის შერჩევა შესაძლოა აღმოჩნდეს გასაღები ამოცანის გადაჭრაში. იგივე ითქმის პარალელური ამოცანის შე,თხვევაში. თუ მონაცემთა სტრუქტურაზე წვდომა ხორციელდება სხვადასხვა დინებიდან, ან იგი სრულიად უცვლელი უნდა იყოს რომ მისი მონაცემები არასოდეს იცვლებოდეს და სინქრონიზებაც არ იყოს საჭირო, ან ან პროგრამის აგებულება უნდა უზრუნველოყოფდეს რომ ცვლილებები სწორადაა სინქრონიზებული დინებებს შორის. ერთი გზა არის დავგეგმოთ მონაცემთა სტრუქტურა ერთდროული წვდომებისთვის. ამ დროს შეგვიძლია გამოვიყენოთ მუტექსები და პირობიანი ცვლადები. ამის რამდენიმე მაგალითი უკვე ვნახეთ.

ეს თავი იწყება რამდენიმე ზოგადი სახელმძღვანელო რჩევით მონაცემთა სტრუქტურის აგებისთვის. შემდეგ მუტექსებს და პირობით ცვლადებს გამოვიყენებთ სტეკის, რიგის და უფრო რთული მონაცემთა სტრუქტურების ასაგებად.

რას ნიშნავს ერთდროულობისთვის სტრუქტურის აგება?

საძირკვლის დონეზე, მონაცემთა სტრუქტურის აგება ერთდროულობისთვის ნიშნავს, რომ მრავალ დინებას შეუძლია ერთდროული წვდომა სტრუქტურაზე, მასზე ერთი და იმავე ან განსხვავებული მოქმედებების შესრულება, და ყოველი დინება დაინახავს დაცულ ინვარიანტებს, არავითარი მონაცემები არ დაიკარგება და არავითრი რბოლის ვითარება არ შეიქმნება. ასეთი სტუქტურა იწოდება დინება-უსაფრთხო. ზოგადად, მონაცემთა სტრუქტურა შესაძლოა დავუდი იყოს ერთდროული წვდომის გარკვეული სახისთვის, მაგალითად თუ რამდენიმე დინება სხვადასხვა საქმეს აკეთებს, შესაძლოა ეს უსაფრთხო იყოს, მაგრამ რამდენიმე დინება ერთი და იმავე მოქმედებით იწვევდეს გართულებას.

ერთდროულობაზე გამიზნული დაგეგმარება ნიშნავს უფრო მეტს: ეს ნიშნავს რომ დინებებს, რომლებიც აღწევენ მონაცემთა სტრუქტურაში, ერთდროულობამ შეუქმნას ხელსაყრელი პირობები. მუტექსის ბუნება ისეთია, რომ იგი გამორიცხავს ორმხრივობას და მრავალმხრივობას, მხოლოდ ერთ დინებას შეუზლია დროის გარკვეულ შუალედში მუტექსის ჩაკეტვა. შესაბამისად, მუტექსი იმის ხარჯზე იცავს მონაცემთა სტრუქტურას, რომ თავიდან იცილებს ჭეშმარიტად ერთდროულ წვდომას მის მიერ დაცულ მონაცემებზე.

ამას ეწოდება **სერიალიზება**: დინებები რიგ-რიგობით აღწევენ მუტექსით დაცულ სტრუქტურაში. საჭიროა გულმოდგინების და ღრმა განსჯის ჩადება მონაცემთა სტრუქტურის აგებულებაში, რომ მივღწიოთ ჭეშმარიტად ერთდროულ წვდომას. ერთი სახის მონაცემთა სტრუქტურებს აქვთ მეტი შესაძლებლობა ჭეშმარიტი ერტდოულობისთვის ვიდრე სხვებს, მაგრამ ნებისმიერ შემთხვევაში იდეა ერთია: რაც უფრო მცირეა დაცული არე, მით უფრო ნაკლები მოქმედება სერიალიზდება და დიდია ერთდროულობის შესაძლებლობა.

<<< სახელმძღვანელო ერთდროულობისთვის მონაცემთა სტრუქტურის დასაგეგმარებლად

განვიხილოთ ერთდროულობისთვის მონაცემთა სტრუქტურის დაგეგმარების ორი ასპექტი: უზრუნველვყოთ მონაცემთა სტრუქტურის უსაფრთხოება და მივაღწიოთ ჭეშმარიტად ერთდროულ წვდომას. წინა თავებში ახსნილი იყო, თუ როგორ გავაკეთოთ დინება-უსაფრთხო მონაცემთ სტრუქტურა:

- დარწმუნდით, რომ ვერცერთი დინება ვერ ნახავს მონაცემთა სტრუქტურას მდგომარეობაში, როდესაც ინვარიანტები გატეხილია სხვა დინების მოქმედების შედეგად;
- იზრუნეთ დასწრების ვითარების გამორიცხვაზე, რაც დამახასიათებელია მონაცემთა სტრუქტურების ინტერფეისისთვის. ეს გააკეთეთ ფუნქციების შემუშავებით მთლიანი მოქმედებებისთვის და არა მოქმედებების ნაწილებისთვის.
- ყურადღება მიაქციეთ, როგორ იქცევა მონაცემთა სტრუქტურა გამონაკლისის არსებობის შემთხვევაში რომ თავი დავიზღვიოთ გატეხილი ინვარიანტებისგან.
- მონაცემთა სტრქუტურის გამოყენებისას შეძლებისდაგვარად შეამცირეთ ჩიხის შესაძლებლობა ჩაკეტვების ხილვადობის შემცირებით და ჩალაგებული ჩაკეტვებისთვის თავის არიდებით სადაც ეს შესაძლებელია.

მნიშვნელოვანია აგრეთვე, რომ შევზღუდოთ მომხმარებლები. თუ რაიმე დინება აღწევს მონაცემთა სტრუქტურაში რაიმე კერძო ფუნქციით, რომელი ფუნქციაა უსაფრთხო სხვა დინებიდან გამოძახებისთვის?

ეს გადამწყვეტი კითხვაა. ზოგადად, კონსტრუქტორები და დესტრუქტორები თხოულობენ ერთპიროვნულ წვდომას მონაცემთა სტრუქტურაზე, მაგრამ მომხმარებლის პასუხისმგებლობაა იმის უზრუნველყოფა, რომ რომ მათში შეღწევა არ მოხდება ვიდრე აგება და მთავრდება ან მას შემდეგ რაც დაშლა დაიწყო.

მეორე ასპექტი არის ჭეშმარიტად ერთდროული წვდომის უზრუნველყოფა. ამის აწყობისას სასარგებლოა რომ პროგრამისტმა განიხილოს ასეთი კითხვები:

- შეიძლება შეზღუდოს ჩაკეტვის ხილვადობა, რაც ოპერაციის ზოგიერთ ნაწილს საშუალებას მისცემს რომ შესრულდეს ჩაკეტვის გარეთ?
- შესაძლებელია თუ არა მონაცემთა სტრუქტურის სხვადასხვა ნაწილის დაცვა სხვადასხვა მუტექსით?
- ყველა ოპერაციას დაცვის ერთი და იგივე დონე სჭირდება?
- შეიძლება თუ არა რომ მარტივმა ცვლილებამ მონაცემთა სტრუქტურაში გააუმჯობესოს ერთდროულობის შესაძლებლობები ოპერაციული სემანტიკაზე გავლენის გარეშე?

ყველა ამ კითხვას აღძრავს ერთი იდეა: როგორ შეგიძლიათ მინიმუმამდე დაიყვანოთ სერიალიზებების რაოდენობა, რომელიც უნდა მოხდეს და როგორ შექმნათ მივაღწიოთ ჭეშმარიტი ერთდროულობის უდიდეს რაოდენობას? იშვიათი არაა როდესაც მონაცემთა სტრუქტურებში რამდენიმე დინება უბრალოდ კითხულობს მონაცემებს, მაშინ როდესაც იმ დინებას, რომელმაც უნდა შეცვალოს მონაცემთა სტრუქტურა, უნდა ჰქონდეს ერთპიროვნული წვდომა. ეს ხდება ისეთი კონსტრუქციების გამოყენებით, როგორიცაა std :: shared_mutex. ამის მსგავსად, საკმაოდ გავრცელებულია როდესაც მონაცემთა სტრუქტურა მხარს უჭერს დინებებიდან ერთდროულ წვდომას როდესაც ისინი ასრულებენ სხვადასხვა მოქმედებას, მაშინ როდესაც ასერიულებენ დინებებს რომლებიც აკეთებენ ერთნაირ მოქმედებებს.

<<< ჩაკეტვაზე დაფუძნებული მონაცემთა სტრუქტურები

ჩაკეტვაზე დაფუძნებული მონაცემთა სტრუქტურის დაგეგმარება მდგომარეობს იმის უზრუნველყოფაში, რომ საჭირო მუტექსი იკეტება სტრუქტურაში შეღწევის დროს და ჩაკეტვა გრძელდება დროის აუცილებელი მინიმუმის განმავლობაში. ეს საკმაოდ ძნელია ერთი მუტექსის გამოყენების შემთხვევაში. ამ დროს გიწევთ ყურადღების მიქცევა რომ მუტექსის დაცვის გარეშე მონაცემებზე არ მოხდეს წვდომა და დასწრების ვითარებაც არ იქმნება. თუ იყენებთ განსხვავებულ მუტექსებს სტრუქტურის განსხვავებული ნაწილების დასაცავად, ეს საკითხები

უფრო რთულდება, ახლა ჩიხის საფრთხეც ჩნდება. ასე რომ სტრუქტურის ინტერფეისი კიდევ უფრო მეტი ყურადღებითაა შესასწავლი რამდენიმე მუტექსის შემთხვევაში, ვიდრე ერთის დროს.

<<< დინება-უსაფრთხო სტეკი ჩაკეტვებით

სტეკი მარტივიმონაცემთა სტრუქტურაა. იყენებს ერთ მუტექსს. ორი კითხვა ჩნდება მასთან დაკავშირებით: არის თუ არა დინება-უსაფრთხო? რა ხდება ჭეშმარიტი ერთდროულობის მიღწევის თვალსაზრისით?

შემდეგ ლისტინგში აღდგენილია დინება-უსაფრთხო სტეკი წინა მასალიდან. მიზანია `td::stack<>`-ის მსგავსი მონაცემთა სტრუქტურის შექმნა, რომელსაც შეუძლია მონაცემების ჩაყრა სტეკში და მათი უკან ამოყრაც.

ლისტინგი 4.1. დინება-უსაფრთხო სტეკის განსაზღვრა

```
#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(std::move(new_value));                ←❶
	}
	std::shared_ptr<T> pop()
	{
		std::lock_guard<std::mutex> lock(m);
		if (data.empty()) throw empty_stack();   ←❷
		std::shared_ptr<T> const res(
		     std::make_shared<T>( std::move(data.top())));    ←❸
		data.pop();      ←❹
		return res;
	}
	void pop(T& value)
	{
		std::lock_guard<std::mutex> lock(m);
		if (data.empty()) throw empty_stack();
		value = move(data.top());      ←❺
		data.pop();              ←❻
	}
```

```
        bool empty() const  {
              std::lock_guard<std::mutex> lock(m);
              return data.empty();
        }
};
```

რიგრიგობით შვხედოთ თითოეულ სახელმძღვანელო პრინციპს და ვნახოთ თუ როგორაა ისინი აქ გამოყენებული.

პირველ რიგში, როგორც ხედავთ, ძირითადი დინების უსაფრთხოება უზრუნველყოფილია თითოეული წევრი-ფუნქციის დაცვით m მუტექსის ჩაკეტვით. ეს უზრუნველყოფს, რომ მხოლოდ ერთი დინება წვდება მონაცემებს დროის ნებისმიერ მომენტში, ასეა უზრუნველყოფილი თითოეული წევრის ფუნქციის მიერ ინვარიანტების შენაჩუნება. ვერცერთი დინება ვერ ხედავს გატეხილ ინვარიანტს.

მეორე, არსებობს დასწრების ვითარების შესაძლებლობა `empty()`-სა და ერთ-ერთ `pop()` ფუნქციას შორის, მაგრამ იმის გამო, რომ კოდი ცხადად ამოწმებს, შიგა სტეკი არის თუ არა ცარიელი, თან ამ დროს უკავია ჩაკეტვა `pop()` -ზე, ეს დასწრების ვითარება არაა პრობლემური. იმის წყალობით, რომ ამოღებული მნიშვნელობის დაბრუნება ხდება `pop()` ფუნქციის გამოძახების ფარგლებში, ვახერხებთ თავიდან ავიცილოთ შესაძლო დასწრების ვითარება, რასაც ექნებოდა ადგილი განცალკევებული `pop()` და `pop()` წევრი ფუნქციების შემთხვევაში (როგორც ეს ოპერაციები არის `std::stack<>`-ში).

შემდეგ, არსებობს გამონაკლისების რამდენიმე შესაძლო წყარო. მუტექსის ჩაკეტვამ შესაძლოა გამოიწვიოს გამონაკლისი, თუმცა ეს არა მხოლოდ უკიდურესად იშვიათია (რადგან მიუთითებს მუტექსთან ან სისტემურ მარაგებთან დაკავშირებულ გართულებას), ის ასევე პირველი მოქმედებაა ნებისმიერ წევრ-ფუნქციაში. რადგან არავითარი მონაცემი არ შეცვლილა, ეს უსაფრთხოა. მუტექსის გახსნა ვერ ჩავარდება, ამიტომ ესაც უსაფრთხოა და `td::lock_guard<>`-ის გამოყენება მუტექსი არასოდეს დაგვრჩება ჩაკეტილი.

`data.push` -ის ❶ გამოძახებამ შესაძლოა გამოისროლოს გამონაკლისი თუ ან მონაცემის მნიშვნელობის ან ასლირება ან გადაადგილება გამოისვრის გამონაკლისს, ან თუ საკმარისი მეხსიერების გამოყოფა ვერ მოხერხდა ქვემდებარე (შიგა) მონაცემთა სტრუქტურისთვის. ნებისმიერ შემთხვევაში `std::stack<>` უზრუნველყოფს რომ ეს იქნება უსაფრთხო.

`pop()`-ის პირველ გადატვირთვაში, კოდს შეუძლია `empty_stack` ❷ გამონაკლისის გამოსროლა. მაგრამ არაფერი შეიცვლება და ესაც უსაფრთხოა. შედეგის შექმნა ❸ -მა შესაძლოა გამოისროლოს გამონაკლისი, რამდენიმე მიზეზის გამო: `std::make_shared`-ს შეუძლია გამოსროლა როცა არ შეუძლია მეხსიერების გამოყოფა ახალი ობიექტისთვის, ან დასაბრუნებელი მონაცემების ასლის ან გაადგილების კონსტრუქტორებს შეუძლიათ გამოსროლა როდესაც ასლს აკეთებენ ან გაადგილებელ ახლად გამოყოფილ მეხსიერებაში. ორივე შემთხვევაში, C++ და სტრანდარტული ბიბლიოთეკა უზრუნველყოფენ რომ ადგილი არ ექნება მეხსიერების გაჟონვას და ახალი ობიექტი (თუ არის ასეთი) სწორად დაიშლება. რადგან თქვენ *ჯერ კიდევ* არ გაქვთ შეცვლილი ქვემდებარე სტეკი, ყველაფერი წესრიგშია. `data.pop` -ის ❹ გამოძახება უეჭველი არ გამოისვრის, ისევე როგორც შედეგის დაბრუნება, ამიტომ `pop`-ის ეს გადატვირთვა გამონაკლის-უსაფრთხოა.

`pop()`-ის მეორე გადატვირთვა მსგავსია, იმის გამოკლებით რომ ამჯერად ასლით მინიჭება ან გაადგილებით მინიჭება უფრო მეტადაა ❺ გამოსროლის შეძლო წყარო, ვიდრე ახალი ობიექტის ან `std::shared_ptr`-ის ნიმუშის აგება. ისევ, თქვენ არ ცვლით მონაცემთა სტრუქტურას `data.pop()`-ის ❻

გამოძახებამდე, რაც კვლავ გარანტირებულია რომ არ ისვრის, ამიტომ ეს გადატვირთვაც გამონაკლის-უსაფრთხოა.

დასასრულ, `empty()` არ ცვლის რაიმე მონაცემს, ამიტომ არის გამონაკლის-უსაფრთხო.

აქ არის ჩიხის ერთი-ორი შესაძლებლობაც, რადგან იძახებთ მომხმარებლის მიერ შექმნილ კოდს ჩაკეტილ ნაკვეთში: ასლის ან გადაადგილების კონსტრუქტორს (❶,❸), აგრეთვე ასლით მინიჭების და გაადგილებით მინიჭების ❺ ოპერატორს სტეკში მოთავსებულ მონაცემებზე. აგრეთვე, შესაძლოა მომხმარებლის მიერ შექმნილ `new` ოპერატორს. თუ ეს ფუნქციები იძახებენ სტეკის წევრ-ფუნქციებს რომლებიც ჩასამატებელ ან ამოსაღებ მონაცემებს კეტავენ რამენაირად, მაშინ როდესაც უკვე იდო ჩაკეტვა წევრი ფუნქციების გამოძახების მომენტში, მაშინვე ჩნდება ჩიხის შესაძლებლობა. მაგრამ გონივრულია მოვითხოვოთ რომ სტეკის მომხმარებლები იყვნენ ამაზე პასუხისმგებლები. არაა გონივრული ვიფიქროთ რომ ჩავსვამთ ან ამოვიღებთ მონაცემს სტეკიდან ყოველგვარი ასლის ან მეხსიერების გამოყოფის გარეშე.

რადგან ყველა წევრი-ფუნქცია იყენებს `std::lock_guard<>`-ს მონაცემების დასაცავად, ამიტომ უსაფრთხოა დინებების ნებისმიერი რაოდენობის მიერ სტეკის წევრი -ფუნქციების გამოძახება. ერთადერთი წევრი-ფუნქციები, რომლების არაა უსაფრთხო, არის კონსტრუქტორი და დესტრუქტორი, მაგრამ ესაც არ იწვევს გართულებას: ობიექტი იგება და იშლება მხოლოდ ერთხელ. არაა კარგი აზრი წევრი-ფუნქციების გამოძახება სანახევროდ აგებული ან სანახევროდ წაშლილი ობიექტისთვის, გინდ სერიულად, გინდ ერთდროულად. მომხმარებელი მაინც პასუხისმგებელია, რომ დინებებს არ ჰქონდეთ საშუალება სანახევროდ აგებული ან დაშლილი ობიექტის წევრების გამოძახების.

როგორც ვნახეთ, მხოლოდ ერთ რომელიმე დინებას შეუძლია დროის ნებისმიერ მომენტში სტეკის მონაცემებში შეღწევა. დინებების ასეთი გასერიულება ქმნის დანართის წარმადობის შეზღუდვის შესაძლებლობას როდესაც არის საკმაო ჭიდილი სტეკისთვის: როდესაც დინება ელოდება სტეკს, მას არ შეუძლია რაიმე სასარგებლოს საქმის შესრულება. იმისათვის რომ მოიცადოს, დინებამ გარკვეული შუალედების შემდეგ უნდა გამოიძახოს `empty()`, ან `pop()` და დაიჭიროს `empty_stack` გამონაკლისები. შემდეგ მაგალითში ნაჩვენებია თუ როგორ შეიძლება ამ მოცდების ჩართვს მონაცემთა სტრუქტურაში პირობიანი ცვლადის გამოყენებით.

<<< დინება-უსაფრთხო რიგი ჩაკეტვებით და პირობიანი ცვლადებით

შემდეგ ლისტინგში აღდგენილია ადრე განხილული დინება-უსაფრთხო რიგის კოდი. იგი აგებულია `std::queue<>` კონტეინერის ადაპტერის საფუძველზე. ინტერფეისი გადამუშავებულია რათა სტრუქტურა იყოს უსაფრთხო რამდენიმე დინებიდან ერთდროული წვდომის შემთხვევაში.

ლისტინგი 4.2. დინება-უსაფრთხო რიგის სრული კლასი პირობიანი ცვლადით

```
template<typename T>
class threadsafe_queue
{
private:
	mutable std::mutex mut;
	std::queue<T> data_queue;
	std::condition_variable data_cond;
public:
	threadsafe_queue()
	{}
	void push(T new_value)
	{
		std::lock_guard<std::mutex> lk(mut);
		data_queue.push(std::move(new_value));
		data_cond.notify_one();                    ←❶
	}
	void wait_and_pop(T& value)          ←❷
	{
		std::unique_lock<std::mutex> lk(mut);
```

```
            data_cond.wait(lk, [this] {return !data_queue.empty(); });
            value = std::move(data_queue.front());
            data_queue.pop();
        }
        std::shared_ptr<T> wait_and_pop()     ←❸
        {
            std::unique_lock<std::mutex> lk(mut);
            data_cond.wait(lk, [this] {return !data_queue.empty(); });    ←❹
            std::shared_ptr<T> res(
                std::make_shared<T>(std::move(data_queue.front())));
            data_queue.pop();
            return res;
        }
        bool try_pop(T& value)
        {
            std::lock_guard<std::mutex> lk(mut);
            if (data_queue.empty())
                return false;
            value = std::move(data_queue.front());
            data_queue.pop();
            return true;
        }
        std::shared_ptr<T> try_pop()
        {
            std::lock_guard<std::mutex> lk(mut);
            if (data_queue.empty())
                return std::shared_ptr<T>();       ←❺
            std::shared_ptr<T> res(
                std::make_shared<T>(std::move(data_queue.front())));
            data_queue.pop();
            return res;
        }
        bool empty() const
        {
            std::lock_guard<std::mutex> lk(mut);
            return data_queue.empty();
        }
};
```

აქ ნაჩვენები რიგის განხორციელება მსგავსია სტეკის იმპლემენტაციის, განსხვავებაა `data_cond.notify_one()` ❶ `push()`-ში და `wait_and_pop()` ფუნქციები ❷ და ❸. `try_pop()`-ის ორი გადატვირთვა თითქმის იმეორებს ლისტინგ 4.1-ის `pop()` ფუნქციას. განსხვავება ისაა რომ ესენი არ ისვრიან გამონაკლისს როდესაც რიგი ცარიელია. სამაგიეროდ, ისინი ან აბრუნებენ `bool` ტიპის მნიშვნელობას იმის საჩვენებლად მნიშვნელობის ამოღება მოხერხდა თუ არა, ან ნულპოინტერს როდესაც ვერანაირი მნიშვნელობა ვერ ამოიღება პოინტერის დამაბრუნებელი გადატვირთვით. თუ არ ჩავთვლით `wait_and_pop()` ფუნქციებს, სტეკისთვის ჩატარებული ანალიზი აქაც ზუსტად ისევე გამოდგება.

ახალი `wait_and_pop()` ფუნქციები ჭრიან სტეკის შემთხვევაში აღნიშნულ ამოცანას, რაც შეეხებოდა ამოსაღები მონაცემისთვის დალოდებას. ნაცვლად იმისა რომ უწყვეტად ვუშვათ `empty()`, მომლოდინე დინებას შეუძლია გაუშვას `wait_and_pop()` და მონაცემთა სტრუქტურა დაამუშავებს მოცდის საკითხს პირობიანი ცვლადით. `data_cond.wait()` -ის გამოძახება არ დაბრუნდება, ვიდრე ქვემდებარე რიგში არ იქნება ერთი ელემენტი მაინც. აღარაა გართულება ცარიელ რიგთან და მონაცემები კვლავ დაცულია მუტექსის ჩაკეტვით. ეს ფუნქციები არ ქმნიან ახალ შესაძლებლობას ჩიხისთვის და ინვარიანტებიც შენარჩუნებულია.

მაგრამ ახალ ფუნქციებს სჭირდებათ მეტი ყურადღება უსაფრთხოების კუთხით. როდესაც რამდენიმე დინება ელოდება მონაცემის შეყვანას რიგში, მხოლოდ ერთი იქნება გაღვიძებული

`data_cond.notify_one()`-ით. მაგრამ თუ ეს დინება გამოისვრის გამონაკლისს `wait_and_pop()`-ში, მაგალითად ახალი `std::shared_ptr<>`-ის ❹ აგებისას, შემდეგ არცერთი დინება აღარ გაიღვიძებს. თუ ეს მიუღებელია, გამოძახება შეიძლება შეიცვალოს `data_cond.notify_all()`-ით, რაც გააღვიძებს ყველა დინებას, მაგრამ მათი უმეტესობა დაიძინებს ისევ როდესაც ნახავს ცარიელ რიგს. მეორე არადანი არის `wait_and_pop()` -ში `notify_one()`-ის გამოძახება გამონაკლისის გამოსროლის შემთხვევაში, რომ სხვა დინებან სცადოს შენახული მნიშვნელობის ამოღება. მესამე არადანია `std::shared_ptr<>`-ის ინიცილიზების გადატანა `push()`-ში და `std::shared_ptr<>`-ის ნიმუშის შენახვა უშუალოდ მონაცემის ნაცვლად. შემდეგ ქვემდებარე რიგიდან `std::shared_ptr<>`-ის ასლირება არ გაისვრის გამონაკლისს და `wait_and_pop()` კვლავ დაცული არის. შემდეგი ლისტინგი გვიჩვენებს რიგის იმპლემენტაციას ამის გათვალისწინებით:

ლისტინგი 4.3. დინება-უსაფრთხო რიგი **`std::shared_ptr<>`**-ის ნიმუშების ფლობით

```
template<typename T>
class threadsafe_queue
{
private:
	mutable std::mutex mut;
	std::queue<std::shared_ptr<T> > data_queue;
	std::condition_variable data_cond;
public:
	threadsafe_queue()
	{}
	void wait_and_pop(T& value)
	{
		std::unique_lock<std::mutex> lk(mut);
		data_cond.wait(lk, [this] {return !data_queue.empty(); });
		value = std::move(*data_queue.front());	←❶
		data_queue.pop();
	}
	bool try_pop(T& value)
	{
		std::lock_guard<std::mutex> lk(mut);
		if (data_queue.empty())
			return false;
		value = std::move(*data_queue.front());	←❷
		data_queue.pop();
		return true;
	}
	std::shared_ptr<T> wait_and_pop()
	{
		std::unique_lock<std::mutex> lk(mut);
		data_cond.wait(lk, [this] {return !data_queue.empty(); });
		std::shared_ptr<T> res = data_queue.front();		←❸
		data_queue.pop();
		return res;
	}
	std::shared_ptr<T> try_pop()
	{
		std::lock_guard<std::mutex> lk(mut);
		if (data_queue.empty())
			return std::shared_ptr<T>();
		std::shared_ptr<T> res = data_queue.front();	←❹
		data_queue.pop();
		return res;
	}
	void push(T new_value)
	{
		std::shared_ptr<T> data(
```

```
                std::make_shared<T>(std::move(new_value)));        ←❺
            std::lock_guard<std::mutex> lk(mut);
            data_queue.push(data);
            data_cond.notify_one();
        }
        bool empty() const
        {
            std::lock_guard<std::mutex> lk(mut);
            return data_queue.empty();
        }
};
```

`std::shared_ptr<>`-ის საშუალებით მონაცემების ფლობის ძირიდი შედეგები სწორხაზოვანია: `pop()` ფუნქციები, რომლებიც იღებენ ცვლადს რეფერენსით ახალი მნიშვნელობის მისაღებად, ახლა უწევთ შენახული პოინტერის განმისამართება - ❶ და ❷. `pop()` ფუნქციები, რომლებიც აბრუნებენ `std::shared_ptr<>`-ის ნიმუშს, შეუძლიათ მისი ამოღება რიგიდან, ❸ და ❹, გამომძახებელთან დაბრუნების წინ.

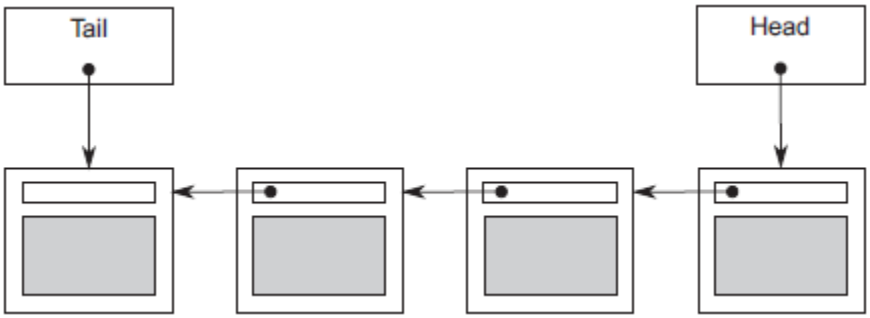
`std::shared_ptr<>`-ის მიერ მონაცემის ფლობას აქვს დამატებითი სარგებელი: ახალი ნიმუშის განთავსება (allocation) ახლა შესაძლოა განხორციელდეს `push()`-ში ჩაკეტვის გარეთ ❺, მაშინ როდესაც წინა ლისტინგში ეს კეთდებოდა ჩაკეტვის ფლობის დროს და `pop()`-ში. რადგან მეხსიერების გამოყოფა ზოგადად საკმაოდ ძვირი ოპერაციაა, ასეთი ცვლილება სასარგებლოა რიგის წარმადობისთვის - მცირდება მუტექსის ფლობის დროს, რაც სხვა დინებებს აძლევს ამ შუალედში რიგზე მოქმედებების განხორციელების შესაძლებლობას.

ისევე როგორც სტეკის შემთხვევაში, ერთი მუტექსის გამოყენება მთლიანი მონაცემთა სტრუქტურის დასაცავად ზღუდავს რიგის მიერ მხარდაჭერილ ერთდროულობას - დროის ნებისმიერ ერთ მომენტში მხოლოდ ერთი დინება აკეთებს რაღაცა საქმეს რიგზე. ამ შეზღუდვის ნაწილი მოდის იმპლემენტაციაში `std::queue<>`-ს გამოყენებისგან. სტანდარტული კონტეინერის გამოყენებით გაქვთ ერთი მთლიანი მონაცემი რაც ან დაცულია ან არა. იმისათვის რომ განახორციელოთ უფრო წვრილად დაქუცმაცებული ჩაკეტვები და მიაღწიოთ უფრო მაღალი დონის ერთდროულობას, საჭიროა მონაცემთა სტრუქტურის იმპლემენტაციის სრულად მართვა.

<<< დინება-უსაფრთხო რიგი წვრილად დაქუცმაცებული ჩაკეტვებით და პირობიანი ცვლადებით

ჩავიხედოთ რიგის იმპლემენტაციაში და შევეცადოთ რომ მის განსხვავებულ ნაწილებს დავადოთ განსხვავებული მუტექსები.

რიგისთვის უმარტივესი მონაცემთა სტრუქტურა არის ბმული სია. რიგი შეიცავს `head` პოინტერს, რომელიც უთითებს რიგის პირველ წევრს, ხოლო რიგის ყოველი წევრი უტითებს მის მომდევნო წევრზე. წევრი მონაცემების ამოღება რიგიდან ხდება `head` პოინტერის ჩანაცვლებით მისი მომდევნო წევრის პოინტერით და მონაცემის დაბრუნებით ძველი `head` -იდან.



რიგს წევრები ემატება მეორე ბოლოში. ამის გასაკეთებლად, რიგი შეიცავს `tail` პოინტერს, რომელიც უთითებს სიის ბოლო ელემენტს. ახალი ელემენტები ემატება ბოლო კვანძის `next` პოინტერის დაყენებით ახალი კვანძის მისამართზე და შემდეგ `tail` პოინტერის განახლება მისი ახალ კვანძზე მიმართვით. როდესაც რიგი ცარიელია, ორივე, `head` და `tail` არის ნულპოინტერი.

შემდეგი ლისტინგი გვიჩვენებს რიგის მარტივ იმპლემენტაციას, დაფუძნებულს ლისტინგ 4.2-ის გაკრეჭილი რიგის ინტერფეისზე. აქ არის მხოლოდ ერთი `try_pop()` და არცერთი `wait_and_pop()`, რადგან ეს რიგი განკუთვნილია მხოლოდ ერთ-დინებიანი მოხმარებისთვის.

ლისტინგი 4.4. მარტივი ერთდინებიანი რიგის იმპლემენტაცია

```
template<typename T>
class queue
{
private:
	struct node
	{
		T data;
		std::unique_ptr<node> next;
		node(T data_) : data(std::move(data_)) {}
	};
	std::unique_ptr<node> head;		←❶
	node* tail;				←❷
public:
	queue() : tail(nullptr)  {}
	queue(const queue& other) = delete;
	queue& operator=(const queue& other) = delete;
	std::shared_ptr<T> try_pop()
	{
		if (!head)
		{
			return std::shared_ptr<T>();
		}
		std::shared_ptr<T> const res(
			std::make_shared<T>(std::move(head->data)));
		std::unique_ptr<node> const old_head = std::move(head);
		head = std::move(old_head->next);		←❸
		if (!head)
			tail = nullptr;
		return res;
	}
	void push(T new_value)
	{
		std::unique_ptr<node> p(new node(std::move(new_value)));
		node* const new_tail = p.get();
		if (tail)
			tail->next = std::move(p);		←❹
		else
			head = std::move(p);		←❺
		tail = new_tail;		←❻
	}
};
```

პირველ რიგში, კოდი იყენებს `std::unique_ptr<>`-ს, რადგან ეს უზრუნველყოფს რომ ისინი (და ის მონაცემები რომლებსაც ისინი უთითებენ) წაიშლება მაშინვე როდესაც ისინი მეტად აღარაა საჭირო. საკუთების ეს ჯაჭვი იმართება `head` -იდან. `tail` არის ნედლი პოინტერი, რადგან ის უთითებს იმ კვანძს, რომელსაც უკვე ფლობს ერთ-ერთი ეული პოინტერი.

თუ ამ კოდის გადატანას დავაპირებთ მრავალდინებიან გარემოში, რამდენიმე საკითხი გამოიწვევს გართულებებს წვრილად-დაქუცმაცებული ჩაკეტვების გამოყენებისას. რადგან გაქვთ ორი წევრი `head` ❶ და `tail` ❷, არსებითად შეგიძლიათ გამოიყენოთ ორი მუტექსი, ერთი `head` -ის და მეორე `tail` -ის დასაცავად. ვნახოთ რა შეიძლება გართულდეს.

მთავარი და ცხადი ისაა, რომ `push()`-ს შეუძლია ორივეს, `head` -ის ❺ და `tail` -ის ❻ შეცვლა. ამიტომ მოუწევს ორივე მუტესის ჩაკეტვა. ეს არაა გადაუჭრელი სირთულე, რადგან ორი

მუტექსის ჩაკეტვა შესაძლებელია. სახიფათო გართულება ისაა, რომ როგორც `push()` ასევე `pop()` აღწევენ კვანძის `next` პოინტერში: `push()` განაახლებს `tail->next` -ს ❹, ხოლო `try_pop()` კითხულობს -ს `head->next` ❸. თუ რიგში არის ცალი ელემენტი, მაშინ `head==tail`, ამიტომ ორივე, `head->next` და `tail->next` არის ერთი და იგივე ობიექტი, რაც საჭიროებს დაცვას. რადგა არ შეგვიძლია არის თუ არა იგივე ობიექტი თუ არ წავიკითხავთ როგორც `head` -ს ასევე `tail` -ს, ამიტომ გვიწევს ერთი და იგივე მუტექსის ჩაკეტვა როგორც `push()`-ში ასევე `try_pop()`-ში. ამგვარად საქმე არაა იმაზე უკეთესად ვიდრე ეს იყო ადრე.

<<< ერთდროულობის ხელშეწყობა მონაცემების განცალკევების გზით

ახლა ხელით გავაკეთოთ სიაზე დაფუძნებული მარტივი რიგი. გავუკეთოთ ცრუ (ფიქტიური, `dummy`) თავი. ასე ერთი კვანძი მაინც ყოველთვის იქნება სიაში. ცარიელი რიგისთვის, `head` და `tail`, ორივე მიუთითებს ცრუ კვანძზე, იმის ნაცვლად რომ იყვნენ NUL. ეს კარგია, რადგან `try_pop()` აღარ შედის `head->next`-ში როდესაც რიგი ცარიელია. თუ დაამატებთ რიგში კვანძს (ისე რომ ერთი ნამდვილი კვანძი იყოს), მაშინ `head` და `tail` მიუთითებენ განსხვავებულ კვანძებზე და არაა შეჯიბრი `head->next` და `tail->next` -ზე. გაუარესებაც არის, საჭიროა მონაცემის ზიარი პოინტერით შენახვა, რის გამოც კოდი უფრო მეტად ირიბი ხდება.

ლისტინგი 4.5. მარტივი რიგი ცრუ კვანძით

```
template<typename T>
class queue
{
private:
	struct node
	{
		std::shared_ptr<T> data;		←❶
		std::unique_ptr<node> next;
	};
	std::unique_ptr<node> head;
	node* tail;
public:
	queue() : head(new node), tail(head.get()) {}	←❷
	queue(const queue& other) = delete;
	queue& operator=(const queue& other) = delete;
	std::shared_ptr<T> try_pop()
	{
		if (head.get() == tail)		←❸
			return std::shared_ptr<T>();

		std::shared_ptr<T> const res(head->data);	←❹
		head = move(head->next);		←❺
		return res;				←❻
	}
	void push(T new_value)
	{
		std::shared_ptr<T> new_data(
			std::make_shared<T>(std::move(new_value)));	←❼
		std::unique_ptr<node> p(new node);		←❽
	❾→	tail->data = new_data;
		tail->next = std::move(p);
		tail = tail->next.get();
	}
};
```

❸-ში ერთმანეთს დარდება `head` და `tail`, იმისგასარკვევად არის თუ არა რიგი არაცარიელი. რადგან თავი ეული პოინტერია, ამიტომ საჭიროა **`head.get()`**-ის გამოძახება. რადგან ახლა კვანძში მონაცემი პოინტერის საშუალებით ინახება, ამიტომ შეგვიძლია პირდაპირ პოინტერის

ამოღება ❹ - საჭირო აღარაა T ტიპის ახალის ნიმუშის შექმნა. დიდი სხვაობა არის push()-ში: პირველ რიგში უნდა გავაკეთოთ T ტიპის ახალის ნიმუში გროვაში და და მისი საკუთრების უფლება გადასცეთ std::shared_ptr<T>-ში ❼ (შევნიშნოთ. რომ std::make_shared<>-ის გამოყენება თავიდან გვაცილებს მეორედ მეხსიერების გამოყოფას რეფერენსების მთვლელისთვის). ახალი კვანძი გამიზნულია რომ გახდეს ახალი ცრუ კვანძი. ძველი ცრუ კვანძი გახდება ბოლო რეალური კვანძი, ამიტომ მის მონაცემს აკავშირებთ new_value-სთან ❾. ცრუ კვანძი არის სულ პირველი რაც იქმნება, ამიტომ ეს ხდება კონსტრუქტორში ❷.

ვნახოთ, თუ რა გაკეთდა დინება-უსაფრთხო რიგის შესაქმნელად. push() ახლა უკვე მხოლოდ tail-თან ურთიერთობს და არა head-თან. try_pop() ორივესთან ურთიერთობს. თუმცა, tail მხოლოდ საწყის შედარებაში საჭიროა. შესაბამისი ჩაკეტვა ხანმოკლეა. ცრუ კვანძი ხელს უწყობს რომ push() და try_pop() არასოდეს მუშაობენ ერთსა და იმავე კვანძზე.

მიზანი არის ერთდროულობისთვის მაქსიმალური შესაძლებლობების შექმნა, ამიტომ საჭიროა რომ ჩაკეტვები გრძელდებოდეს რაც შეიძლება ცოტა ხანს. push() ადვილია, ჩაკეტვა საჭიროა tail -თან ყველა წვდომაზე. ❽ და ❾-ს შორის.

try_pop() არაა ასე მარტივი. პირველ რიგში, უნდა ჩავკეტოთ head და ჩაკეტვა დავიჭიროთ ბოლომდე. ამავე დროს, ამ ჩაკეტვის განმავლობაში უნდა ჩავკეტოთ ხანმოკლე დროით tail (შესაბამისი მუტექსით) იმ მიზნით რათა გავარკვიოთ, არის თუ არა ცრუ კვანძის გარდა სხვა კვანძიც სიაში.

ლისტინგი 4.6. დინება-უსაფრთხო რიგი წვრილად დაქუცმაცებული ჩაკეტვებით

```
template<typename T>
class threadsafe_queue
{
private:
	struct node
	{
		std::shared_ptr<T> data;
		std::unique_ptr<node> next;
	};
	std::mutex head_mutex;
	std::unique_ptr<node> head;
	std::mutex tail_mutex;
	node* tail;
	node* get_tail();
public:
	threadsafe_queue() : head(new node), tail(head.get()) {}
	threadsafe_queue(const threadsafe_queue& other) = delete;
	threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

	std::shared_ptr<T> try_pop();
	void push(T new_value);
};
template<typename T>
typename threadsafe_queue<T>::node* threadsafe_queue<T>::get_tail()
{
	std::lock_guard<std::mutex> tail_lock(tail_mutex);
	return tail;
}
template<typename T>
std::shared_ptr<T> threadsafe_queue<T>::try_pop()
{
	head_mutex.lock();
	if (head.get() == get_tail())
	{
		head_mutex.unlock();
		return std::shared_ptr<T>();
```

```
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        head_mutex.unlock();
        return old_head->data;}
}
template<typename T>
void threadsafe_queue<T>::push(T new_value)
{
        std::shared_ptr<T> new_data(
              std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = tail->next.get();
}
```

შევხედოთ კოდს კრიტიკულად ზემოთ მოყვანილი სახელმძღვანელო პრონციპების თვალსაზრისით. ინვარიანტების დაცულობასთან დაკავშირებით უნდა სრულდებოდეს:

- `tail->next==nullptr.`
- `tail->data==nullptr.`
- `head.get()== tail` ნიშნავს რომ სია ცარიელია
- ერთელემენტიან სიას აქვს `head->next.get()==tail.`
- ნებისმიერი `x`-ისთვის სიაში, `x.get()!=tail`, `x->data` მიუთითებს `T`-ს ობიექტზე და `x->next` მიუთითებს სიაში მომდევნო კვანძზე. `x->next.get()==tail` ნიშნავს რომ `x` ბოლოა სიაში.
- `head` -იდან დაწყებული, `next` კვანძების მიყოლა საბოლოოდ მიგვიყვანს `tail`-მდე.

`push()` სწორხაზოვანია. `tail_mutex` იცავს ცვლილებებს და მონაცემებიც სწორად ყენდება.

`try_pop()`-ში, საჭიროა `tail_mutex` -ის გამოყენება `tail`-ის დასაცავად. ასევე აუცილებელია თავი დავიცვათ მონაცემების რბოლისგან `head` -იდან მონაცემები სკითხვის დროს. თუ არ ვიქონიებთ შესაბამის მუტექსს, შესაძლებელი იქნება ერთი დინების მიერ `try_pop()`-ის გამოძახება და სხვა დინების მიერ ერთდროულად `push()`-ის გამოძახება. არ იქნება განსაზღვრელი რიგი მათ გამოძახებებში. იმ შემთხვევაშიც კი, თუ თითოეული ფუნქცია იჭერს ჩაკეტვას მუტექსზე, ისინი იჭერენ ჩაკეტვებს *განსხვავებულ* მუტექსებზე და შესაძლოა შეაღწიონ ერთი და იმავე მონაცემებზე. საბედნიეროდ, `tail_mutex` -ის ჩაკეტვა `get_tail()`-ში ჭრის ყველა პრობლემას. `get_tail()`-ში კეტავს იგივე მუტექსს რასაც `push()`, არსებობს განსაზღვრული შედეგი ამ ორი ფუნქციის გამოძახებას შორის. ან `get_tail()` სრულდება `push()`-ის წინ, რა შემთხვევაშიც `get_tail()` ხედავს `tail`-ის ძველ მნიშვნელობას, ან იგი სრულდება `push()`-ის შემდეგ და ხედავს ახალ მნიშვნელობას.

არსებითია, რომ `get_tail()`-ის გამოძახება სრულდება `head_mutex` -ის ჩაკეტვის შიგნით. წინააღმდეგ შემთხვევაში, მაგალითად,

```
node* const old_tail = get_tail();
std::lock_guard<std::mutex> head_lock(head_mutex);
if (head.get() == old_tail)
```

. . .

სცენარის შემთხვევაში, შესაძლოა პირველ და მეორე სტრიქონს შორის შეიცვალოს როგორც `head` ასევე `tail`. ამ დროს, უკვე ჩაკეტვაში, შესაძლოა არა მხოლოდ `tail` უკვე არ იყოს სიის ბოლო (`tail`), არამედ შესალოა იგი აღარც იყოს სიის ნაწილი, საერთოდ. ამ დროს მესამე სტრიქონში მოყვანილი შედარება ჩავარდება თუნდაც `head` იყოს ერთადერთი კვანძი სიაში. შესაბამისად, როდესაც განვაახლებთ `head` -ს, იგი (ახალი თავი) აღმოჩნდება `tail`-ის შემდეგ, ანუ სიის გარეთ და ერთ-ერთი ინვარიანტი ირღვევა.

გამონაკლისების საკითხი უფრო საინტერესოა. try_pop()-ში ერთადერთი მოქმედება, რომელსაც შეუძლია გამონაკლისების გამოსროლა, არის მუტექსის ჩაკეტვები, და მონაცემები არ შეიცვლება თუ მუტექსი არ ჩაიკეტა. ამიტომ try_pop() არის გამონაკლის-უსაფრთხო. მეორე მხრივ, push() გამოყოფს მეხსიერებას T-ს ახალი ნიმუშისთვის გროვაში და node-ის ახალი ნიმუშისთვის, ორიცეს შეუძლია გამონაკლისის გამოსროლა. მაგრამ ორივე ახალშექმნილი ნიმუში მიენიჭება ჭკვიან პოინტერებს, ამიტომ ისინი თავისუფლდებიან გამონაკლისების გამოსროლის შემთხვევაში. მას შემდეგ რაც ჩაკეტვა მოთხოვნილია, არცერთ სხვა მოქმედებას push() -ში არ შეუძლია გამონაკლისის გამოსროლა. ამიტომ push() აგრეთვე არის გამონაკლის-უსაფრთხო.

რადგან ინტერფეისი არ შეგვიცვლია, ამიტომ ჩიხისთვის არ არსებობს რაიმე გარე შესაძლებლობა. არც შინაგანი შესაძლებლობები არსებობს; მხოლოდ ერთგანაა ორივე მუტექსი ერთდროულად გამოყენებული , მაგრამ იქაც ყოველთვის ჯერ იკეტება head_mutex, შემდეგ tail_mutex.

ბოლო საკითხი არის ერთდროულობა. განხილულ სტრუქტურას საგრძნობლად მეტი შესაძლებლობა აქვს ერთდროულობისთვის, ვიდრე ლისტინგ 4.2-ში განხილულს, რადგან ჩაკეტვები არის წვრილად დაქუცმაცებული და უფრო მეტი კეთდება ჩაკეტვის საზღვრების გარეთ. მაგალითად, push()-ში ახალი კვანძის და მონაცემის წევრისთვის მეხსიერება გამოიყოფა ჩაკეტვის გარეშე. ეს ნიშნავს, რომ მრავალ დინებას შეუძლია იგივე საქმის კეთება ერთდროულად. მართალია, მხოლოდ ერთ დინებას შეუძლია სიაში ახალი კვანძის დამატება, მაგრამ ამის გასაკეთებელი კოდი შედგება რამდენიმე მარტივი პოინტერის მინიჭებისგან. შესაბამისად, ჩაკეტვა არ ჩერდება დიდხანს.

ასევე, try_pop() იჭერს tail_mutex მუტექსს მხოლოდ მცირე ხნის განმავლობაში, რათა tail -იდან წაკითხვა დაიცვას. შესაბამისად, try_pop() თითქმის მთლიანად შესაძლოა push()-ის ერთდროულად შესრულდეს. ასევე, head_mutex-ის ჩაკეტვის ქვეშ მხოლოდ მოქმედებების მინიმალური რიცხვი სრულდება. ძვირი დაშლა (delete) სრულდება ჩაკეტვის გარეთ node-ის პოინტერის დესტრუქტორში.

<<< წევრისთვის მოცდა სიიდან ამოსაღებად

დინება-უსაფრთხო რიგი წვრილად დაქუცმაცებული ჩაკეტვებით ჯერ მხოლოდ try_pop()-ს გვთავაზობს, და ისიც ერთ გადატვირთვას. განვიხილოთ wait_and_pop()-ის შექმნის საკითხი. ეს მოითხოვს push()-ის გადაკეთებასაც ბოლოში data_cond.notify_one()-ის დამატებით, რაც ადვილია. თუმცა, ეს ბოლო სტრიქონი არ უნდა მოჰყვეს ჩაკეტვაში.

wait_and_pop()-ში წინასწარ უნდა ჩამოვყალიბდეთ, თუ სად მოვიცადოთ, პრედიკატი როგორი იქნება და რომელი მუტექსი ჩაიკეტება. პირობა, რომელსაც ველოდებით, არის „რიგი არაცარიელია“, ანუ head!=tail. მაგრამ ამას სჭირდება ორივე მუტექსის ჩაკეტვა. თუმცა უკვე ვიცით რომ tail_mutex-ის ჩაკეტვა საკმარისია tail-ის წაკითხვისთვის, მაგრამ უშუალოდ შედარებისთვის - არა. თუ პრედიკატი იქნება head!=get_tail(), თქვენ გჭირდებათ მხოლოდ head_mutex-ის დაჭერა. იგივე ჩაკეტვის გამოყენება შეგიძლიათ data_cond.wait()-ის გამომახებაზე. დანარჩენი, იგივეა რაც try_pop()-ის.

try_pop()-ის მეორე გადატვირთვა და შესაბამისი wait_and_pop()გადატვირთვა კარგ დაფიქრებას საჭიროებს. value = std::move(*head->data) ჩაკეტვის მოქმედების ქვეშ უნდა მოვახვედროთ, წინაარმდეგ შემთხვევაში გართულება წარმოიქმნება ამ მინიჭების გამო გამოსროლილი გამონაკლისის შემთხვევაში.

შემდეგი ლისტინგი გვიჩვენებს დინება-უსაფრთხო რიგის კოდს C++ -ის ჩვეულ სტილში.

ლისტინგი 4.6. დინება-უსაფრთხო რიგი წვრილად დაქუცმაცებული ჩაკეტვებით

```
template<typename T>
class threadsafe_queue
{
```

```
private:
	struct node
	{
		std::shared_ptr<T> data;
		std::unique_ptr<node> next;
	};
	std::mutex head_mutex;
	std::unique_ptr<node> head;
	std::mutex tail_mutex;
	node* tail;
	std::condition_variable data_cond;
	std::unique_lock<std::mutex> wait_for_data();
	node* get_tail();
public:
	threadsafe_queue() :head(new node), tail(head.get()) {}
	threadsafe_queue(const threadsafe_queue& other) = delete;
	threadsafe_queue& operator=(const threadsafe_queue& other) = delete;
	std::shared_ptr<T> try_pop();
	bool try_pop(T& value);
	std::shared_ptr<T> wait_and_pop();
	void wait_and_pop(T& value);
	void push(T new_value);
	bool empty();
};

template<typename T>
void threadsafe_queue<T>::push(T new_value)
{
	std::shared_ptr<T> new_data(
		std::make_shared<T>(std::move(new_value)));
	std::unique_ptr<node> p(new node);
	{
		std::lock_guard<std::mutex> tail_lock(tail_mutex);
		tail->data = new_data;
		tail->next = std::move(p);
		tail = tail->next.get();
	}
	data_cond.notify_one();
}

template<typename T>
typename threadsafe_queue<T>::node* threadsafe_queue<T>::get_tail()
{
	std::lock_guard<std::mutex> tail_lock(tail_mutex);
	return tail;
}

template<typename T>
std::unique_lock<std::mutex> threadsafe_queue<T>::wait_for_data()
{
	std::unique_lock<std::mutex> head_lock(head_mutex);
	data_cond.wait(head_lock, [&] {return head.get() != get_tail(); });
	return std::move(head_lock);
}

template<typename T>
std::shared_ptr<T> threadsafe_queue<T>::wait_and_pop()
{
	std::unique_lock<std::mutex> head_lock(wait_for_data());
	data_cond.wait(head_lock, [&] {return head.get() != get_tail(); });
	std::unique_ptr<node> old_head = std::move(head);
	head = std::move(old_head->next);
```

```
        head_lock.unlock();
        return old_head->data;
}

template<typename T>
void threadsafe_queue<T>::wait_and_pop(T& value)
{
        std::unique_lock<std::mutex> head_lock(wait_for_data());
        data_cond.wait(head_lock, [&] {return head.get() != get_tail(); });
        value = std::move(*head->data);
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        head_lock.unlock();
}

template<typename T>
std::shared_ptr<T> threadsafe_queue<T>::try_pop()
{
        head_mutex.lock();
        if (head.get() == get_tail())
        {
                head_mutex.unlock();
                return std::shared_ptr<T>();
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        head_mutex.unlock();
        return old_head->data;
}

template<typename T>
bool threadsafe_queue<T>::try_pop(T& value)
{
        head_mutex.lock();
        if (head.get() == get_tail())
        {
                head_mutex.unlock();
                return false;
        }
        value = std::move(*head->data);
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        head_mutex.unlock();
        return true;
}

template<typename T>
bool threadsafe_queue<T>::empty()
{
        std::lock_guard<std::mutex> head_lock(head_mutex);
        return (head.get() == get_tail());
}
```

<<< სავარჯიშოები