# დინებების მართვა



**რა არის ერთდროულობა (concurrency)?** ძალიან გამარტივებულად, ერთდროულობა არის ორი ან მეტი ერთმანეთისგან გამიჯნული მოქმედება, რომლებიც ერთდროულად მიმდინარეობს.

კომპიუტერებთან დაკავშირებით ერთდროულობა გვესმის როგორც ერთი ცალკეული სისტემა, რომელიც ასრულებს რამდენიმე ერთმანეთისგან დამოუკიდებელ საქმიანობას პარალელურად. ე.ი. არა მიმდევრობით, ანუ ერთი მეორის მიყოლებით. ეს საკმაოდ გავრცელებული რამაა. მაგალითად, მრავალამოცანიანი ოპერაციული სისტემები საშუალებას აძლევს ერთ სამაგიდო კომპიუტერს რომ მან ერთდროულად რამდენიმე დანართი (application) ამუშაოს ამოცანებს შორის გადართვების საშუალებით. ასევე, ძლიერი სერვერები მრავალი პროცესორით, რაც ჭეშმარიტი ერთდროულობის შესაძლებლობას იძლევა. ამჟამად, სამაგიდო კომპიუტერებიც იძლევა ჭეშმარიტი ერთდროულობის შესაძლებლობას (და არა ერთდროულობის ილუზიას). როდესაც კომპიუტერი ქმნის ორი ამოცანის ერთდროული გადაჭრის ილუზიას ამოცანებს შორის გადართვის საშუალებით, იგი უფრო მეტ დროს ხარჯავს ვიდრე ამ ორი ამოცანის მიმდევრობით შესრულებაზე. მართლაც, თითოეული ამოცანის გარკვეული ულუფის შესრულების შემდეგ, ვიდრე გადაერთვება მეორეზე, სისტემამ უნდა შეინახოს CPU-ს მდგომარეობა და მიმთითებელი მიმდინარე ამოცანის გარკვეულ ინსტრუქციებზე. და ასე ყოველ ჯერზე გადართვის დროს.

როგორიც არ უნდა იყოს კომპიუტერი (მრავალპროცესორიანი, მრავალბირთვიანი...), ერთდროულობისთვის არსებითია აპარატურული დინებების რაოდენობა (number of hardware threads).

ერთდროულობის გამოსაყენებლად არსებობს ორი ძირითადი მიდგომა.

<<< **ერთდროულობა მრავალჯერადი პროცესებით** (with multiple processes). ეს ნიშნავს დანართის დაყოფას მრავლობით, ერთმანეთისგან განცალკევებულ, ერთ-დინებიან პროცესებად, რომლებიც მუშაობენ ერთი და იმავე დროს. მაგალითად, როგორც ჩვეულებრივ ვამუშავებთ ვებ-დამთვალიერებელს (web browser) და ტექსტურ რედაქტორს ერთი და იმავე დროს, ერთდროულად. ამ განცალკევებულ პროცესებს შეუძლიათ გზავნილების გაცვლა ერთმანეთს შორის პროცესთაშორისი სხვადასხვა არხის გამოყენებით (სიგნალები, სოკეტები, ფაილები და სხვა). მიდგომის ნაკლი ისაა, რომ ასეთი ურთიერთობა რთულიცაა, ნელიცაა და ზოჯერ ორივეც ერთად. ეს იმიტომ, რომ ოპერაციული სისტემები ზრუნავენ პროცესების დაცვაზე, რომ ერთმა პროცესმა არ შეცვალოს მეორის მონაცემები, მაგალითად. გარდა ამისა, მრავლობითი პროცესების გაშვება დაკავშირებულია დამატებით ხარჯებთან (overheads): პროცესების გაშვებას სჭირდება დრო, ოპერაციულმა სისტემამ უნდა გამოყოს შინაგანი მარაგები პროცესებისთვის და სხვა.

<<< **ერთდროულობა მრავალჯერადი დინებებით.** დინებები გვანან შედარებით მცირე პროცესებს: თითოეული მათგანი ეშვება დანარჩენებისგან დამოუკიდებლად და თითოეულს

შეუძლია ინსტურქციების განსხვავებული მიმდევრობის გაშვება. მაგრამ ერთი და იმავე პროცესის ყველა დინება იზიარებს მისამართების ერთი და იმავე სივრცეს და მონაცემების უმეტესობა მიღწევადია ყველა დინებიდან. გლობალური ცვლადები გლობალურად რჩება, ხოლო ობიექტების პოინტერების და რეფერენსების გაცვლა შესაძლებელია დინებებს შორის. პროცესებს შორისაც არის შესაძლებელი მეხსიერების გაზიარება, მაგრამ ეს ბევრად რთულია დინებების შემთხვევასთან შედარებით, რადგან ერთი და იმავე მონაცემების მისამართები მეხსიერებაში განსხვავებულია განსხვავებული პროცესებისთის.

დინებებისთვის საზიარო მისამართების სივრცე და მონაცემების შემცირებული დაცვა სხვადასხვა დინებებიგან ამცირებს ზედნადებს დინებების გამოყენების შემთხვევაში პროცესების გამოყენების შემთხვევისგან. თუმცა ამას აქვს მეორე მხარე - საზიარო მონაცემების ცვლილება და ნახვა დამატებით ღონისძიებებს საჭიროებს.

**<<< ერთდროულობა და პარალელიზმი.** ამ ორ ცნებას დიდი თანაკვეთა აქვს. ორივე ეხება ერთდროულად მრავალი დინების გაშვებას. თუმცა პარალელიზმი უფრო წარმადობაზეა მოგეზილი. ერთდროულობა ნიშნავს, რომ ძირითად ამოცანას წარმოადგენს პასუხისმგებლობების ან ქვეამოცანების გამიჯვნა. მაგალითად, განვიხილოთ DVD-ის შემსრულებელი დანართი სამაგიდო კომპიუტერისთვის. ასეთ დანართს ძირითადად აქვს ორი სახის პასუხისმგებლობა: უნდა წაიკითხოს გაშიფროს და გაუშვას გრაფიკა და ბგერა, და უნდა მიიღოს მითითებები მომხმარებლისგან - ვთვათ, მენიუს ნახვა, ან შესრულების შეწყვეტა. ერთი დინებისთვის, დანართს მოუწევს მომხმარებლის მითითების (თუ არის ასეთი) გადამოწმება დროის გარკვეული შუალედების შემდეგ. მრავალი დინების შემთხვევაში, ერთი დინება მოუვლის მოწყობილობას, მეორე იზრუნებს მომხმარებლის მითითებებზე.

**<<< ამოცანების და მონაცემების პარალელიზმი.** ამოცანების პარალელიზმი ნიშნავს ერთი ამოცანის დახლეჩას რამდენიმე ამოცანად და მათ გაშვებას პარლელურად. დახლეჩილი ამოცანები შესაძლოა ალგორითმის სხვადასხვა ნაწილს ასრულებდნენ, ან შესაძლებელია რომ ყველა ასრულებდეს ერთი და იმავე მოქმედებებს, ოღონდ განსხვავებულ მონაცემებზე - ეს ბოლო მიდგომა არის მონაცემების პარალელიზმი.

**<<< ორი ერთდროული დინების პირველი მაგალითი.** განვიხილოთ კარგად ნაცნობი პროგრამა, რომლითაც, როგორც წესი, იწყება ყველა ახალი თემის შესწავლა პროგრამირებაში - პროგრამა „გამარჯობა, სამყარო“. მისი ერთდინებიანი სახე არის:

```
#include <iostream>
int main()
{
	std::cout << "Hello World\n";
}
```

მისი ერთდროული ვარიანტი არის:

```
#include <iostream>
#include <thread>          // #1
void hello()                    //  #2
{
	std::cout << "Hello Concurrent World\n";
}
int main()
{
	std::thread t(hello);   //#3
	t.join();                    //  #4
}
```

პირველი განსხვავება არის #1, გასაგებია რასაც აკეთებს. შემდეგ, გზავნილის დაბეჭდვა გადატანილია ცალკე ფუნქციაში - #2. ეს იმიტომ, რომ ყოველ დინებას უნდა ჰქონდეს საწყისი ფუნქცია, რომელშიც (thread of execution) აღმასრულებელი დინება დაიწყება. მოცემულ

დანართში, საწყისი დინებისთვის ასეთი ფუნქცია არის `main()`. ყველა სხვა დანარჩენი დინებისთვის, ასეთი ფუნქცია განსაზღვრული უნდა იყოს `std::thread` ობიექტის კონსტრუქტორში. ჩვენს შემთხვევაში #3. მას შემდეგ რაც ახალი დინება გაეშვება, საწყისი დინება აგრძელებს შესრულებას. თუ იგი არ დაელოდება ახალი დინების დასრულებას, მაშინ პროგრამა დამთავრდება და ახალ დინებას არ მიეცემა დამთავრების საშუალება. ამის გამო არის #4 სტრიქონი ასეთი.

**<<< რა ღირს ახალი დინების გაშვება?** როგორც აღვნიშნეთ, ახალი დინების გაშვება ზედნადებ ხარჯებთანაა დაკავშირებული. მიკროწამებში ამის გაზომვა შეგვიძლია შემდეგი მარტივი კოდის გამოყენებით:

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

void func() {}
int main()
{
	thread t;
	auto st = chrono::high_resolution_clock::now();

	t= thread (func);
	t.join();
	auto diff = chrono::high_resolution_clock::now() - st;
	auto time = chrono::duration_cast<chrono::microseconds>(diff);
	cout << time.count() << endl;
}
```

**<<< დინების გაშვება.** C++ -ის ყოველ პროგრამას აქვს სულ მცირე ერთი დინება, რომელიც გამოიძახებს main() პროგრამას. ახალი დინების გაშვება იწყება `std::thread` ობიექტის აგებით, რომელიც განსაზღვრავს დინებაში გასაშვებ ამოცანას. მარტივ შემთხვევაში, ეს ამოცანა შესაძლოა იყოს ფუნქცია void მნიშვნელობით, რომლის დამთავრების შემდეგ დინება გაჩერდება. უფრო რთულ შემთხვევებში, გასაშვები ამოცანა შესაძლოა იყოს ფუნქციური ობიექტი, რომლის სხვადასხვა ნაწილის შესრულება და დინების გაჩერება უნდა მოხდეს გზავნილების რაიმე სისტემით. ვნახოთ ორივეს მაგალითი:

```
void some_func();
std::thread my_thread(some_func);
```

ხოლო გამოძახებადი ობიექტისთვის:

```
class background_task
{
public:
	void operator()() const
	{
		some_func1();
		some_func2();
	}
};
background_task f;
std::thread my_thread(f);
```

აქ, გადაწოდებული ობიექტი ასლდება ახალი დინების საცავში, საიდანაც ხდება მისი გამოძახება. შეგვიძლია დროებითი ობიექტის გადაწოდებაც, თუმცა ამ დროს ყურადღებაა საჭირო, რომ დინების ნაცვლად არ მივიღოთ ფუნქციის განაცხადი

```
std::thread my_thread(background_task());
```

რომელიც ქმნის ფუნქციას ერთადერთი პარამეტრით (უპარამეტრო ფუნქციის პოინტერი). იმის მისაღებად, რაც სინამდვილეში გვინდა, უნდა ავიღოთ შემდეგი ორიდან ერთ-ერთი განაცხადი:

```
std::thread my_thread((background_task()));
std::thread my_thread{ background_task() };
```

აღნიშნულ პრობლემას აგრეთვე ადვილად ვაღწევთ თავს, თუ დინებაში გასაშვებად გადავაწვდით ლამბდა ფუნქციას. მაგალითად, შემდეგი ფუნქტორის

```
class background_task
{
public:
	void operator()() const
	{
		do_something();
		do_something_else();
	}
};
```

შესაბამისი ობიექტის ნაცვლად, დინებას გადავაწოდოთ  როგორც ანომიმი ლამბდა ფუნქცია, რომელიც იგივე საქმეს აკეთებს:

```
std::thread my_thread([]
(
	do_something();
do_something_else();
});
```

<<< **დინების დასრულება.** ვნახოთ რა მოხდება, თუ გავუშვით დინება, მაგრამ არ ვიზრუნეთ მის დასრულებაზე:

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

void func() { cout << "Hello!" << endl; }
int main()
{
	thread t = thread(func);
	this_thread::sleep_for(std::chrono::seconds(5));
}
```

როგორც ვხედავთ, პროგრამა ავარიულად მთავრდება, მიუხედავად იმისა, რომ ჩვენ ვიზრუნეთ და გამოვყავით საკმარისი დრო ძირითად დინებაში, რომ ფონურ რეჟიმში გაშვებული დინებას დაესრულებინა მუშაობა. აუცილებელი, რომ ცხადად ან არაცხადად ვიზრუნოთ თვით დინების დასრულებაზე.

ცხადად ეს კეთდება ორნაირად: ან უნდა დაველოდოთ მის დასრულებას `join()` მეთოდის გამოყენებით, ან უნდა გამოვცალკევდეთ მისგან (detaching). ამ ბოლო შემთხვევაში, `std::thread` ობიექტი აღარაა დაკავშირებული აქტიურ აღმასრულებელ დინებასთან და ამიტომ არაა შემოერთებადი (joinable). ამიტომ შემდეგი კოდი დაგვიწერს, რომ `assert` -ის არგუმენტი არის მცდარი:

```
#include <iostream>
#include <thread>
#include <assert.h>
using namespace std;

void func() { cout << "Hello!" << endl; }

int main()
{
	std::thread t(func);
```

```
	t.detach();
	assert(t.joinable());
}
```

გარდა გზავნილისა, კიდევ რაღაც ჩანს ტექსტში. თუ კოდის ბოლო სტრიქონს გავაკომენტარებთ, დავინახავთ, რომ ვიდრე ფონურ რეჟიმში გასაშვები დინება განცალკევდებოდა, მანამდე მან ტექსტის ნაწილის დაბეჭდვა მოასწრო მხოლოდ  (თუ თქვენმა კომპიუტერმა მოასწრო და მთლიანად დაბეჭდა მისალმება, მაშინ ტექსტის რაოდენობა ან ზომა ზომა გაზარდეთ).

როდესაც დინებას ვუშვებთ ფონურ რეჟიმში (და არ ველოდებით მის დამთავრებას), უნდა ვიზრუნოთ რომ ყველა მონაცემი, რომლებთანაც დინებას აქვს წვდომა, იყოს გამართულ მდგომარეობაში. აი მაგალითი, როდესაც საქმე ფუჭდება იმის გამო, რომ დინების ფუნქცია რეფერირებს ცვლადზე, რომლის არსებობა დამთავრდა დინების დამთავრებამდე:

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

struct func
{
	int& i;
	func(int& i_) :i(i_) {}
	void operator()()
	{
		for (auto j = 0; j< 500000; ++j)
		{
			++i;                                    // #1
			cout << i << endl;
		}
	}
};
void Uh()
{
	int n = 0;
	func my_func(n);
	std::thread t(my_func);
	t.detach();                                     // #2
}                                                       // #3
int main()
{
	Uh();
	cout << "A little time for background thread" << endl;
}
```

*ლისტინგი 1: ფუნქცია ბრუნდება იმ დროს, როდესაც დინებას კვლავ აქვს წვდომა ლოკალურ ცვლადზე*

პროგრამის ყოფაქცევა განუსაზღვრელია. სხვადასხვა კომპიუტერზე მისი ყოფაქცევა განსხვავებულია. თუმცა, თუ რამდენჯერმე გავუშვებთ, დიდი ალბათობით იგი დაეკიდება, რადგან  Uh() ფუნქციის დამთავრების შემდეგ n ცვლადი წყვეტს არსებობას, შესაბამისად, მისი ცვლილების გამო პრობლემები უნდა შეიქმნას, რადგან მეხსიერებაში არაა ადგილი ამ სახელით. დაკიდებების სიხშირე გაიზრდება, თუ მთავარ ფუნქციას ოდნავ გადავაკეთებთ:

```
int main()
{
	Uh();
	int n = 11;
	cout << "A little time for background thread" << endl;
}
```

ამ ამოცანაში, არსებობს სამი საფრთხე:

#1. წვდომა რეფერენსზე, რომელიც სავარაუდოდ მაწანწალა გახდება

#2. არ ველოდებით დინების დამთავრებას
#3. შესაძლოა ახალი დინება ისევ მუშაობდეს...

ამავე დროს, ვიცით რამდენი სიკეთე ახლავს რეფერენსების გამოყენებას და მასზე უარის თქმა დინებების შემთხვევაშიც ძნელია. პროგრამირებაში არსებობს გამოთქმა (idiom): Resource Acquisition Is Initialization (RAII). ამ მიდგომის მიხედვით, დინების გამოყენება რომ უსაფრთხო გახდეს, ამისთვის იქმნება კლასი, რომელიც იძახებს loin() მეთოდს მის დესტრუქტორში. კლასი, რომელშიც გაუქმებული არის ასლის და მინიჭების კონსტრუქტორები, რეფერენსით მიიღებს დინებას. ამ მიდგომით, ლისტინგი 1-ის თემა შემდეგ განვითარებას იღებს:

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

class thread_guard
{
	std::thread& t;
public:
	explicit thread_guard(std::thread& t_) :
		t(t_)
	{}
	~thread_guard()
	{
		if (t.joinable())
		{
			t.join();
		}
	}
	thread_guard(thread_guard const&) = delete;
	thread_guard& operator=(thread_guard const&) = delete;

};
struct func
{
	int& i;
	func(int& i_) :i(i_) {}
	void operator()()
	{
		for (auto j = 0; j< 50000; ++j)
		{
			++i;
			cout << i << endl;
		}
	}
};
void Uh()
{
	int n = 0;
	func my_func(n);
	std::thread t(my_func);
	thread_guard g(t);
	cout << "What can I do?" << endl;
}

int main()
{
	freopen("data.out", "w", stdout);
	Uh();
}
```

*ლისტინგი 2: RAII იდიომის გამოყენება დინების დასამთავრებლად*

აღსანიშნავია Uh() ფუნქციის მეოთხე (დამატებული) სტრიქონი, რომელიც გაშვებულ დინებას დაარქმევს ახალ სახელს, რომელიც მას არ მისცემს დაშლის საშუალებას ვიდრე არ შემოიერთებს მას (თუ უკვე შემოერთებული არ დახვდა ამ დროს).

**<<< არგუმენტის გადაწოდება დინებისთვის.** აქამდე განვიხილავდით მარტივ შემთხვევებს, როდესაც დინება უშვებდა void ტიპის უპარამეტრო ფუნქციას ან ფუნქტორს. თუ ფუნქციას აქვს პარამეტრები, მაშინ მათი გადაწოდება ხდება აღმასრულებელი დინების აგებისას (კოსტრუქტორში). რამდენიმე განსხვავებული სცენარი არსებობს. ვნახოთ შესაბამისი მაგალითები.

1. უპარამეტრო ფუნქციის მაგალითი ზემოთ ვნახეთ.
2. მარტივად ხდება პარამეტრების ასლების გადაწოდება:

```
void printSum(int a, int b)
{
   std::cout << a + b << std::endl;
}

int main()
{
   std::thread t(printSum,10,33);
   t.join();
}
```

ან, ფუნქტორის შემთხვევაში:

```
struct printSum
{
   void operator()(int a, int b)
   {
          std::cout << a + b << std::endl;
   }

};
int main()
{
   std::thread t(printSum(),10,33);
   t.join();
}
```

თუმცა, ამის მსგავსი შედეგი გვექნება, თუ მთავარ ფუნქციას მიცვემთ სახეს:

```
int main()
{
   printSum func;
   std::thread t(func,10,33);
   t.join();
}
```

შემდეგში, სიმარტივისთვის, მხოლოდ ფუნქციებს გავუშვებთ დინებიდან.

აქაც და შემდეგშიც, ლამბდა ფუნქციების გამოყენება ამარტივებს საქმეს. ჩვენ წინასწარ შეგვიძლია მოვამზადოთ ლამბდა და შემდეგ იგი გადავაწოდოთ აღმასრულებელ დინებას. მაგალითად:

```
void printSum(int a, int b)
{
   std::cout << a + b << std::endl;
}

int main()
{
   auto lm = []() {printSum(10, 33); };
```

```
   std::thread t(lm);
   t.join();
}
```

3. პოინტერი ჩვეულებრივ ცვლადს წარმოადგენს, ამიტომ მის გადაცემაზე არ შევჩერდებით.
4. ვნახოთ როგორ ხდება რეფერენსის გადაწოდება.

```
void printSum(int a, int &b)
{
   std::cout << a + b << std::endl;
}

int main(){
   int n = 15;
   std::thread t(printSum,10,std::ref(n));
   t.join();
}
```

როგორც ვხედავთ, საჭირო გახდა „რეფერენსის შემფუთველის“ (reference wrapper) კონსტრუქტორის გამოძახება.

აქაც შეგვიძლია იგივე ფანდის გამოყენება:

```
int main() {
   int n = 15;
   auto lm = [&]() {printSum(10, n); };
   std::thread t(lm);
   t.join();
}
```

5. განსაკუთრებულია ისეთი შემთხვევები, როდესაც C++ -ს სჭირდება ტიპების გარდაქმნა. ტიპების გარდაქმნას ეს ენა მარტივად აკეთებს, მაგრამ საქმე ისაა, რომ დინება შესაძლოა მანამდე გაეშვას, ვიდრე ტიპის გარდაქმნა დამთავრდება. მაგალითად:

```
void f(int i, std::string const& s);
void oops(int some_param)
{
   char buffer[1024];               // #1
   sprintf(buffer, "%i", some_param);
   std::thread t(f, 3, buffer);     // #2
   t.detach();
}
```

აქ ახალ დინებაში (#2) გადაეწოდება buffer-პოინტერი ლოკალურ ცვლადეზე (#1). თუ ვერ მოესწრო buffer-ის კონვერტირება სტრინგად, ფუნქცია დამთავრდება და მივიღებთ ავარიულ გარემოებას.

გამოსავალი მდგომარეობს ამ ცვლადის ცხად გარდაქმნაში, მანამდე, ვიდრე გადავაწოდებთ დინებას:

```
void f(int i, std::string const& s);
void not_oops(int some_param)
{
   char buffer[1024];
   sprintf(buffer, "%i", some_param);
   std::thread t(f, 3, std::string(buffer));
    t.detach();
}
```

6. ცალკე თემაა მხოლოდ გადაადგილებადი არგუმენტის გადაწოდება (რომლის გაასლება არ შეიძლება). ასეთებია განსაკუთრებული პოინტერი, თვით დინებები და ზოგიერთი სხვა. ამ დროს, საკმარისია ობიექტზე გამოვიძახოთ `std::move()`:

```
void process_big_object(std::unique_ptr<big_object>);
std::unique_ptr<big_object> p(new big_object);
p->prepare_data(42);
std::thread t(process_big_object, std::move(p));
```

7. გავცელებული შემთხვევაა წევრი ფუნქციების გადაწოდება. ამ დროს, უნდა მივუთითოთ ფუნქცია სრული მისამართით (კლასის სახელის ჩათვლით), ამის შემდეგ კი გადავაწოდოთ ობიექტის მისამართი დამატებით არგუმენტად. მაგალითად:

```
struct printSum
{
   void operator()(int a, int b)
   {
         std::cout << a + b << std::endl;
   }
};
int main()
{
   printSum x;
   std::thread t(&printSum::operator(), &x, 10,55);
   t.join();
}
```

აქაც შეგვიძლია ლამბდა ფუნქციის ფანდი გამოვიყენოთ:

```
int main()
{
   printSum x;
   auto lm = [&]() {x(10, 55); };
   std::thread t(lm);
   t.join();
}
```

8. შედარებით ნაკლებად გავრცელებული შემთხვევაა, როდესაც დინების გამოძახება ხდება კლასის მეთოდში. ამ დროს უნდა მივუთითოთ პოინტერი გამომძახებელ ობიექტზე (this). შემდეგი მარტივ (და ხელოვნურ) მაგალითში კლასს შეუძლია დაბეჭდოს თავისი მონაცემი ძირითად დინებაშიც და ფონურ დინებაშიც. მთავარ პროგრამაში, ვიყენებთ ბეჭდვას ფონურ რეჟიმში.

```
class sample
{
public:
   sample(int k):n(k){ }
   ~sample() { my_thread.join(); }
   void start() { my_thread = std::thread(&sample::printData, this); }
   void printData() { std::cout << n << std::endl; }
private:
   std::thread my_thread;
   int n;
};

int main()
{
   sample x(77);
   x.start();
}
```

<<< დინებაზე საკუთრების უფლების გადაცემა. წარმოვიდგინოთ, რომ გვინდა ისეთი ფუნქციის შექმნა, რომელიც შექმნის დინებას და გაუშვებს მას ზურგში (ფონურ რეჟიმში), მაგრამ გამომძახებელ ფუნქციას უბრუნებს უკან ახალი დინების საკუთრების უფლებას იმის ნაცვლად რომ დაელოდოს მის დამთავრებას. ან პირიქით, გვინდა ახალი დინების შექმნა და მისი საკუთრე-

ბის უფლების გადაცემა რომელიმე ფუნქციისთვის, რომელიც დაელოდება მის დამთავრებას. ორივე შემთხვევაში საჭიროა საკუთრების უფლების გადაცემა ერთი ადგილიდან მეორეზე.

დინება არის გადაადგილებადი ობიექტი. აღმასრულებელი დინების საკუთრების უფლება შესაძლოა გადაადგილდეს `std::thread`-ის ნიმუშებს შორის. შემდეგი მაგალითი გვიჩვენებს ორი აღმასრულებელი დინების შექმნას და მათზე საკუთრების უფლების გადაცემას `std::thread`-ის სამ ნიმუშს შორის:

```
void some_function();
void some_other_function();
std::thread t1(some_function);            // #1
std:: thread t2 = std::move(t1);                  // #2
t1 = std:: thread (some_other_function);  // #3
std:: thread t3;                                  // #4
t3 = std::move(t2);                       // #5
t1 = std::move(t3);                               // #6
```

ახალი დინება იწყება და უკავშირდება `t1` -ს (#1). შემდეგ საკუთრების უფლება გადადის `t2`-ზე მისი აგების მომენტში, მაგრამ ამისთვის საჭიროა `std::move` -ის გამოძახება გადაადგილების განხორციელებისთვის (#2). აღმასრულებელი დინება, რომელიც უშვებს `some_function` ფუნქციას, ახლა დაკავშირებულია `t2`-თან.

შემდეგ, ახალი დინება იწყება და უკავშირდება დროებითი დინებას (#3). `t1`-თან მის დასაკავშირებლად საჭირო არაა `std::move` -ის გამოძახება, რადგან დროებითი ობიექტის გადაადგილება თავისით ხდება არაცხადად.

`T3` იგება ნაგულისხმევი კონსტრუქტორით (#4), რაც ნიშნავს რომ იგი შეიქმნა აღმასრულებელი დინების გარეშე. #5 სტრიქონში, `t2`-თან დაკავშირებულ დინებაზე საკუთრების უფლება გადაეცემა `t3`-ს, კვლავ `std::move` -ის ცხადი გამოძახებით (რადგან `t2` არის სახელის მქონე ობიექტი, ამიტომ მას აქვს საკუთარი მეხსიერება და არაა დროებითი). ამ გადაადგილებების შედეგად, `t1` დაკავშირებულია იმ დინებასთან, რომელიც უშვებს `some_other_function`, `t2`-ს არა აქვს მასთან დაკავშირებული დინება, `t3` დაკავშირებულია იმ დინებასთან, რომელიც უშვებს `some_function`-ს.

ბოლო გადაადგილება (#6) გადასცემს საკუთრების უფლებას `some_function`-ის გამშვებ დინებაზე `t1`-ს, მაგრამ `t1`-თან უკვე არის დაკავშირებული დინებასთან რომელიც უშვებს `some_other_function`-ს. ამიტომ პროგრამის შესაწყვეტად გამოიძახება `std::terminate()`. ეს ხდება `td::thread`-ის დესტრუქტორთან თავსებადობისთვის. ჩვენ ვნახეთ, რომ უნდა დაველოდოთ დინების დამთავრებას, ან გამოვცალკევდეთ მისგან მის დაშლამდე, და ეს წყელაფერი სამართლიანია მინიჭებისთვისაც: ახალი მნიშვნელობის მინიჭების შემდეგ არ შეგვიძლია აღმასრულებელი დინების (წინა მნიშვნელობის) „უპატრონოდ მიგდება“, ისიც უნდა დამთავრდეს.

`td::thread`-ის მიერ გადაადგილების მხარდაჭერა ნიშნავს, რომ საკუთრების უფლება ადვილად გადმოიცემა ფუნქციიდან. მაგალითად, თუ განსაზღვრულია

```
void some_function();
void some_other_function(int);
```

მაშინ შესაძლებელია ფუნქციების შექმნა, რომლებიც შექმნიან აღმასრულებელ დინებებს და დააბრუნებენ მათზე საკუთრების უფლებას:

```
std::thread f()
{
	return std::thread(some_function);
}
ან
std::thread g()
```

```
{
	std::thread t(some_other_function, 42);
	return t;
}
```

`td::thread`-ის მიერ გადაადგილების მხარდაჭერა საშუალებას გვაძლევს, რომ განვავითაროთ ლისტინგ 2-ში აგებული `thread_guard` კლასი, ისე რომ გართულებები აღარ შეიქმნება თუ `thread_guard` ობიექტი უფრო დიდხანს იცოცხლებს ვიდრე ის დინება რომელზეც ის მიუთითებს. ეს იმასაც ნიშნავს, რომ სხვა ვერაფერი შემოიერთებს ან განაცალკევებს დინებს მას შემდეგ, რაც საკუთრება გადაცემული იქნება ობიექტის შიგნით. რადგან ამ ყველაფრის მიზანი არის დინების დამთავრება მისი ხილვადობის არის ამოწურვამდე, ამიტომ კლასს ეწოდება `scoped_thread`. სრული მაგალითი შემდეგია:

```
#include <iostream>
#include <thread>
using namespace std;
class scoped_thread
{
	std::thread t;
public:
	explicit scoped_thread(std::thread t_) :			// #1
		t(std::move(t_))
	{
		if (!t.joinable())					// #2
			throw std::logic_error("No thread");
	}
	~scoped_thread()
	{
		t.join();						// #3
	}
	scoped_thread(scoped_thread const&) = delete;
	scoped_thread& operator=(scoped_thread const&) = delete;
};
struct func
{
	int& i;
	func(int& i_) :i(i_) {}
	void operator()()
	{
		for (auto j = 0; j< 50000; ++j)
		{
			++i;
			cout << i << endl;
		}
	}
};
void f()
{
	int some_local_state{0};
	scoped_thread t{ std::thread(func(some_local_state)) };	//  #4
	cout << "What can I do?" << endl;
}							//  #5
int main()
{
	freopen("data.out", "w", stdout);
	f();
}
```

*ლისტინგი 3: `scoped_thread` და მისი გამოყენების ნიმუში*

წინა ლისტინგისგან განსხვავებით, ახალი (დროებითი) დინება უშუალოდ scoped_thread ობიექტს გადაეცემა (#4). როდესაც ახალი დინება მიაღწევს f() ფუნქციის ბოლოს (#5), scoped_thread ობიექტი დაიშლება და #3-ის ძალით შემოიერთებს #1 კონსტრუქტორის მიერ შექმნილ დინებას. მაშინ როდესაც ლისტინგ 2-ის thread_guard კლასის დესტრუქტორს უწევს იმის შემოწმება, არის დინება შემოერთებადი თუ არა, იგივეს გაკეთება შეიძლება კონსტრუქტორში (#2) გამონაკლისის გამომუშავებით.

td::thread-ის მიერ გადაადგილების მხარდაჭერა უშვებს ისეთი კონტეინერებით სარგებლობას, რომლებიც გადაადგილებას იცნობენ (მსგავსად განახლებული std::vector<>-ისა). ეს ნიშნავს, რომ შეგვიძლია დინებების ვექტორის შექმნა და შემდეგ მათი შემოერთება:

```
void do_work(unsigned id);
void f()
{
	std::vector<std::thread> threads;
	for (unsigned i = 0; i<20; ++i)
	{
		threads.push_back(std::thread(do_work, i));
	}
	std::for_each(threads.begin(), threads.end(),
		std::mem_fn(&std::thread::join));
		Licensed to <null>
}
```

ცხადია, ეს მხოლოდ თვალსაჩინოებისთვის შექმნილი მაგალითია. იგი გვიჩვენებს თუ როგორ შეიძლება დინებების თვითმართვა. იმის ნაცვლად რომ თითოეულ დინებას შევუქმნათ ცალკე ცვლადი და შემდეგ ისინი ცხადად შემოვიერთოთ, დინებებს ვეპყრობით როგორც ერთ ჯგუფს. შემდეგი ნაკვეთი გვიჩვენებს, თუ როგორ შეიძლება დინებების რიცხვის დინამიკურად განსაზღვრა.

<<< დინებათა რაოდენობის განსაზღვრა პროგრამის მსვლელობის დროს. C++ ის სტანდარტული ბიბლიოთეკის ფუნქცია std::thread::hardware_concurrency() აბრუნებს დინებების იმ რაოდენობის მაჩვენებელს, რაც ნამდვილად შეიძლება ერთდროულად გაეშვას პროგრამის მოცემული შესრულების დროს. მრავალბირთვიან სისტემაში ეს შეიძლება იყოს CPU -ს ბირთვების რაოდენობა, მაგალითად. ეს მაინც მხოლოდ მინიშნებაა და ფუნქციამ შესაძლოა დააბრუნოს 0 თუ თუ მონაცემების აღება ვერ მოახერხა. მაინც, იგი სასარგებლოა ამოცანის მცირე ნაწილებად დასახლეჩად.

შემდეგი ლისტინგი (4-ე) წარმოგვიდგენს პარალელური std::accumulate-ის მარტივ ვერსიას. კურსის ბოლო ნაწილში ვნახავთ თუ როგორ შეიძლება სტანდარტული ფუნქციის უკვე არსებული std::reduce ალგორითმი გამოვიყენოთ ამ მიზნით. საკუთარი იმპლემენტაცია გვიჩვენებს რამდენიმე ძირითად იდეას. განვიხილოთ კოდი:

```
template<typename Iterator, typename T>
struct accumulate_block
{
	void operator()(Iterator first, Iterator last, T& result)
	{
		result = std::accumulate(first, last, result);
	}
};
template<typename Iterator, typename T>
T parallel_accumulate(Iterator first, Iterator last, T init)
{
	unsigned long const length = std::distance(first, last);
	if (!length)                                                    // #1
		return init;
	unsigned long const min_per_thread = 25;
```

```
        unsigned long const max_threads =
              (length + min_per_thread - 1) / min_per_thread;            // #2
        unsigned long const hardware_threads =
              std::thread::hardware_concurrency();
        unsigned long const num_threads =                                // #3
              std::min(hardware_threads != 0 ? hardware_threads : 2, max_threads);
        unsigned long const block_size = length / num_threads;           // #4
        std::vector<T> results(num_threads);
        std::vector<std::thread> threads(num_threads - 1);               // #5
        Iterator block_start = first;
        for (unsigned long i = 0; i<(num_threads - 1); ++i)
        {
              Iterator block_end = block_start;
              std::advance(block_end, block_size);                       // #6
              threads[i] = std::thread(                                  // #7
                    accumulate_block<Iterator, T>(),
                    block_start, block_end, std::ref(results[i]));
              block_start = block_end;                                   // #8
        }
        accumulate_block<Iterator, T>()(
              block_start, last, results[num_threads - 1]);              // #9
        std::for_each(threads.begin(), threads.end(),
              std::mem_fn(&std::thread::join));                          // #10
         return std::accumulate(results.begin(), results.end(), init);   // #11
}
```

*ლისტინგი 4: `std::accumulate` -ის სწორხაზოვანი, გულუბრყვილო ვერსია*

თუ დიაპაზონი ცარიელია (#1), უბრალოდ ვაბრუნებთ საწყის მნიშვნელობას `init`. სხვა შემთხვევაში, დიაპაზონსი არის ერთი მაინც ელემენტი და გვიწევს დასამუშავებელი ელემენტების დაყოფა მინიმალური ზომის ბლოკებად და აქედან გამომდინარე, დინებების მაქსიმალური რიცხვის გაგება (#2). ეს იმიტომ, რომ არ გავუშვათ ბევრი დინება მაშინ როდესაც სულ ხუთი ელემენტია დიაპაზონში, რისთვისაც ერთი დინებაც საკმარისია. დინებების რიცხვი არის ამ რაოდენობისად და აპარატურული დინებების რიცხვის მინიმუმი წარმოადგენს გასაშვები დინებების რაოდენობას (#3). აპარატურული დინებების რაოდენობაზე მეტის გაშვება გამოიწვევს გადართვებს და შეანელებს წარმადობას. `std::thread::hardware_concurrency()` თუ გამოძახება დააბრუნებს 0-ს, მასინ უბრალოდ ავიღებთ რაიმე რიცხვს ჩვენი შეხედულებისამებრ. ამ შემთხვევაში აღებული გვაქვს 2. ძალიან ბევრი დინების გაშვება შეანელებს ერთ-ბირთვიან კომპიუტერზე პროგრამის წარმადობას. #3 განსაზღვრავს მონაცემების რაოდენობას ერთი დინებისთვის.

ამ მომენტში ვიცით რამდენი დინება გვჭირდება, ე.ი. შეგვიძლია შევქმნათ ორი ვექტორი: საშუალედო შედეგებისთვის `std::vector<T>`, ხოლო დინებებისთვის `std::vector<std::thread>` (#5). დინება ერთით ნაკლებია, რადგან ამ დროს უკვე მუშაობს ერთი.

დინებების გაშვება ხდება განმეორების შეტყობინებაში: `block_end` იტერატორს მივასწრაფებთ მიმდინარე ბლოკის დასასრულისკენ (#6) და ვუშვებთ ახალ დინებას ამ ბლოკის შედეგების დასაგროვებლად (#7). შემდეგი ბლოკის დასაწყისი არის ამ ბლოკის დასასრული (#8).

მას მერე რაც ყველა დინება გაეშვა, ძირითადი დინება დაამუშავებს ბოლო ბლოკს (#9). ამის შემდეგ `std::for_each` -ის საშუალებით ველოდებით ყველა გაშვებული დინების შემოერთებას (#10). `std::accumulate` –ის ბოლო გამოძახება შეაჯამებს შედეგებს (#11).

ყურადღება მივაქციოთ რამდენიმე გარემოებას. ისეთ `T` ტიპებზე, რომლებზე შეკრების ოპერაცია არ არის ასოციაციური (ნამდვილი რიცხვების ტიპები), `parallel_accumulate` -ის შედეგები შესაძლოა განსხვავდებოდეს `std::accumulate`-ის შედეგებისგან, რადგან შესაკრებები სხვადასხვანაირად დაჯგუფდება.

გამკაცრებულია მოთხოვნები იტერატორებზე. `std::accumulate`-ისთვის საკმარისია ერთჯერადი გავლის *input* იტერატორები, ხოლო მის პარალელურ ვერსიას სჭირდება მრავალჯერადი გავლის იტერატორები, *forward* მაინც.

`T` -ს აგება უნდა იყოს შესაძლებელი ნაგულისხმევი კონსტრუქტორით. სხვანაირად შედეგების ვექტორს ვერ შევქმნით.

ასეთი ცვლილებები ზოგადია პარალელური ალგორითმებისთვის.

**<<< დინების ამოცნობა.** C++ ამ მიზნით იყენებს `std::thread::id` ობიექტებს. მიმდინარე დინების ID-ის მიკვლევა შეიძლება `std::this_thread::get_id()`-ის გამოძახებით. მაგალითად:

```
cout << this_thread::get_id() << endl;
```

შეტყობინებით. შესაძლოა პროგრამამ ყოველ გაშვებაზე სხვადასხვა მნიშვნელობა აჩვენოს, სტანდარტი ამას არ ზღუდავს. თუმცა, ყოველ კონკრეტულ გაშვებაზე, ყოველ გამოძახებაზე იგი ერთი და იმავე მნიშვნელობას გვიჩვენებს. მაგალითად:

```
int main() {
	cout << this_thread::get_id() << endl;
	cout << this_thread::get_id() << endl;
}
```

ანალოგიური ვითარებაა სხვა დინებების შემთხვევაში. მაგალითად:

```
int main()
{
	std::cout << this_thread::get_id() << std::endl;
	std::thread t;
	std::cout << t.get_id() << std::endl;
}
```

რადგან *t*-სთვის აღმასრულებელი დინება შექმნილი არაა, პროგრამა სავარაუდოდ დაბეჭდავს 0-ს, ან რაიმე ისეთ მნიშვნელობას რაც გვიჩვენებს რომ არავითარი დინება არაა.

რისი ტოლია იგი დინების ID-ის? ზუსტი შედეგი დამოკიდებულია იმპლემენტაციაზე, სტანდარტი არ ავალდებულებს კომპილერების შემქმნელებს მის დაზუსტებას. თუმცა, აუცილებლად შესრულდება შემდეგი რამეები: ტოლი დინებები წარმოშობენ ტოლ ID-ებს, ხოლო განსხვავებულები - განსხვავებულებს.

ID-ების გამოყენება ძალიან სასარგებლოა ბევრ შემთხვევაში. მაგალითად, თუ გვინდა რომ პროგრამის გარკვეულ ნაწილში საწყისმა დინებამ, რომელიც უშვებს დანარჩენებს, შეასრულოს სხვებისგან განსხვავებული სამუშაო, შეგვიძლია გამოვიყენოთ ასეთი ტექნიკა:

```
std::thread::id master_thread;
	. . . // master_thread -ში შევინახოთ საწყისი დინების ID
void some_core_part_of_algorithm()
{
	if (std::this_thread::get_id() == master_thread)
	{
		do_master_thread_work();
	}
	do_common_work();
}
```

`std::thread::id` -ის ტიპის ობიექტების ასლის გადაღება და შედარება ნებადართულია. ეს მათ საშუალებას აძლევს რომ გამოყენებულ იქნენ გასაღებებად ასოცირებულ კონტეინერებში.