

# Вишенично програмирање у Це++-у

Стандардна Це++ библиотека

# Нит?

- Нит је програмски ток који се у времену преплиће са извршавањем других програмских токова, тј. других нити.
- Све нити конкуришу за заузеће једног процесора, па се програм који користи нити зове „конкурентан програм“, а писање таквих програма „конкурентно програмирање“.
- Паралелизам је врло сличан конкуренцији, само што има више процесора. Зато је то важна техника.
- У тзв. managed окружењу, main функција се извршава у једној, првој нити.
- У већини програма то је и једина нит.
- Али, из те прве нити могу се створити друге нити.

# Стварање нити

- Од 2011. године, нити су саставни део Це++ стандарда, а стварање и управљање нитима се остварује кроз стандардну библиотеку. Библиотека је урађена по узорну на POSIX нити, само у ООП маниру.
- Стварање нити се обавља стварањем објекта типа `std::thread`.
- Тада се дешава и њено покретање.

```
void print1() {  
    std::cout << "1";  
}
```

```
void main() {  
    std::thread thread1(print1);  
    ...  
}
```

# Стварање нити

```
std::thread thread1 (print1);
```



- Шта може бити параметар?

# Стварање нити

```
std::thread thread1(print1);
```



- Шта може бити параметар?
  - Показивач на функцију (и то смо већ видели)

# Стварање нити

```
struct print1Type {  
    void operator() () {  
        std::cout << "1";  
    }  
};  
  
print1Type print1;  
  
std::thread thread1(print1);
```



- Шта може бити параметар?
  - Показивач на функцију (и то смо већ видели)
  - Функцијски објекат (функтор)

# Стварање нити

```
struct print1 {  
    void operator() () {  
        std::cout << "1";  
    }  
};
```

```
std::thread thread1(print1{});
```



- Шта може бити параметар?
  - Показивач на функцију (и то смо већ видели)
  - Функцијски објекат (функтор)

# Стварање нити

```
std::thread thread1([]() {std::cout << "1"; });
```



- Шта може бити параметар?
  - Показивач на функцију (и то смо већ видели)
  - Функцијски објекат (функтор)
    - Ламбда функција



# Преношење параметара функцији нити

- Функција може примати параметре.

```
struct print {  
    void operator()(std::string s) {  
        std::cout << s;  
    }  
};
```

```
std::thread thread1(print(), "1");  
std::thread thread2(print(), "2");
```

- Без обзира колико је параметара и ког су типа.

```
struct print1 {  
    void operator()(std::string s, int i) {  
        std::cout << s << i;  
    }  
};
```

```
std::thread thread3(print1(), "1", 2);  
std::thread thread4(print2(), "3", 4);
```

# Стварање нити

- Обратити пажњу да ће објекат нити (објекат који представља нит – `std::thread` објекат) бити створен одмах након завршетка конструктора, као и било која друга променљива.
- Сама нит ће у том тренутку бити направљена и у стању приправности, али нема гаранције кад ће бити покренута.

```
void main() {  
    std::thread thread1([]() {std::cout << "1"; });  
    std::cout << "2";  
    ...  
}
```

- Испис може бити и 12 и 21, зависи од тога када ће се нит покренути.

# Уништавање нити

- Погледајмо сада овај код:

```
void main() {  
    std::thread thread1(print1);  
} // thread1 се уништава (позива се деструктор)
```

- Шта се дешава ако нит thread1 није завршила (или чак није ни почела са радом)?
- Сматра се нерегуларним уништавање објекта нити пре него што је нит завршила са радом.
- Због тога је у овом случају потребно да главна нит на неки начин сачека на свршетак извршавања новонастале нити па тек онда да и она сама заврши са радом.

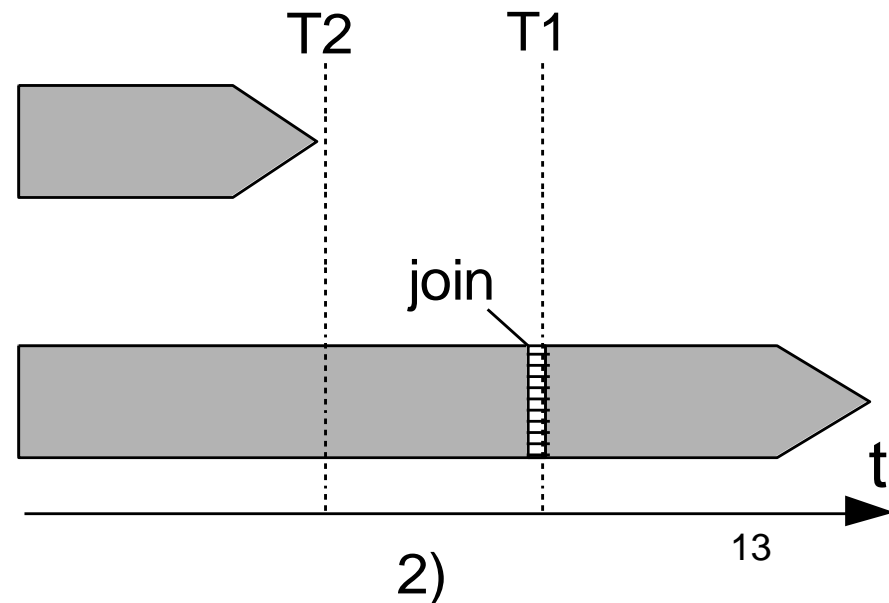
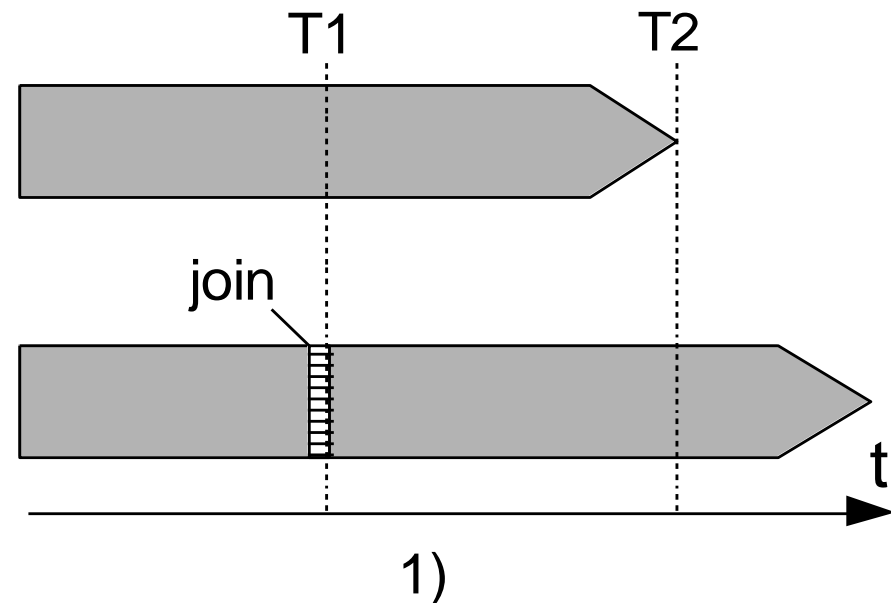
# Прикључивање нити

- Овакав модел синхронизације, у којем једна нит чека на завршетак друге, назива се **прикључивање нит** (енгл. join)
- На овај начин, ток програма који се извршавао у нити која се чека прикључује се току програма нити која чека, и каже се да се једна нит прикључује другој.

```
void main() {  
    std::thread thread1(print1);  
    thread1.join(); // main ће овде чекати све док се thread1  
                   // не заврши  
} // thread1 се уништава (позива се деструктор)
```

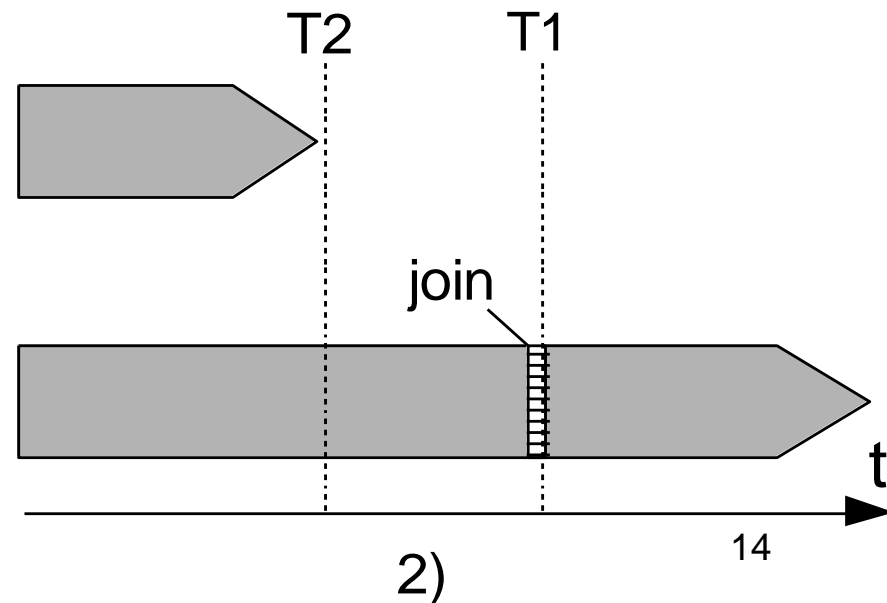
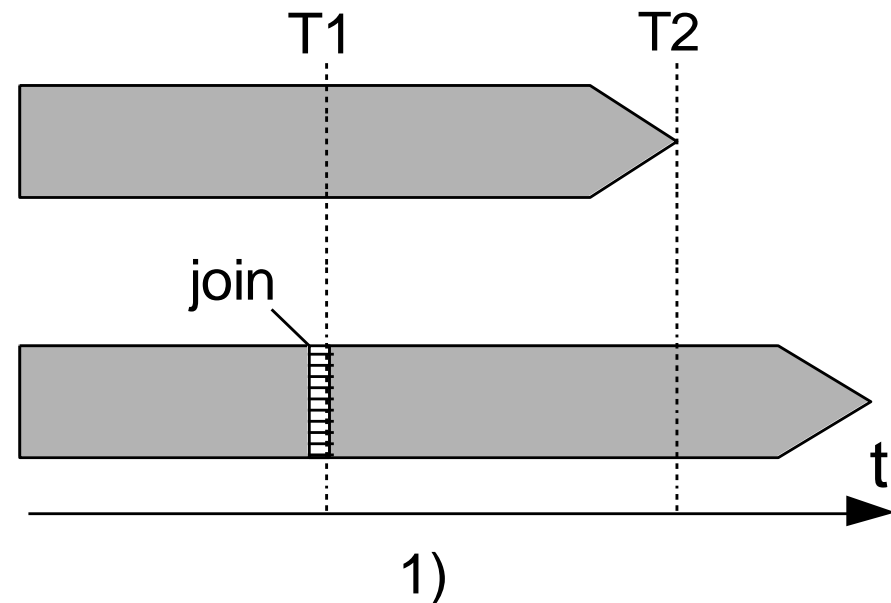
# Прикључивање нити

- Илустрована су два случаја чекања нити на прикључење.
- У првом случају нит којој се прикључује прелази у стање чекања (тренутак T1) пре него што је нит која се прикључује завршила свој ток (тренутак T2).
- У другом случају нит која се прикључује завршава са радом (тренутак T2) пре него што је нит која прикључује спремна да је прикључи (тренутак T1).



# Прикључивање нити

- У првом случају нит којој се прикључује стварно чека неко време.
- У другом случају нит којој се прикључује не чека ништа. Међутим, њу, у меморији, чека информација о завршетку нити која се прикључује.



# Прикључивање нити

- Разрадимо још два сценарија прикључивања нити:
- 1) Нит извршава такву функцију да је не треба прикључивати.
- 2) Нови нит не желимо да прикључимо из нити (блока кода) у којој је настала, већ на неком другом месту.

# Прикључивање нити

- 1) Нит извршава такву функцију да је не треба прикључивати.

```
void foo() {  
    // do something  
}
```

```
void bar() {  
    std::thread thread1(foo);  
    thread1.detach(); // овим саопштавамо да thread1 више није  
                     // прикључива нит  
}
```

- Ресурси неприкључиве нити ће бити аутоматски ослобођени када се нит заврши (јер информација о њеном завршетку се више никога не тиче).
- У принципу, нит може и никада да се не заврши, али то је већ сложена дискусија.



# Прикључивање нити

- 2) Нови нит не желимо да прикључимо из нити (блока кода) у којој је настала, већ на неком другом месту.

```
void foo();

std::thread bar() {
    std::thread thread1(foo);
    return thread1;
}

void main() {
    std::thread mainT{bar()};
    mainT.join();
}
```

```
void foo();

void bar(std::thread t) {
    // ...
    t.join();
}

void main() {
    std::thread thread1(foo);
    bar(std::move(thread1));
}
```

- `std::thread` се понаша слично јединственом показивачу.
- Не може бити два објекта који се односе на исту нит!
- Има само мув конструктор и мув додела.

# Преношење параметара функцији нити, наставак

- Како се преносе параметри функцији нити: по вредности, или по референци?

```
struct foo {  
    void operator() (int x);  
};  
  
std::thread bar() {  
    int i = 6;  
    std::thread t(foo(), i);  
    return t;  
}
```

- Овде је ствар јасна и нема никаквих проблема.

# Преношење параметара функцији нити, наставак

- Како се преносе параметри функцији нити: по вредности, или по референци?

```
struct foo {  
    void operator()(const int& x);  
};
```

```
std::thread bar() {  
    int i = 6;  
    std::thread t(foo(), i);  
    return t;  
}
```

- Када би *i* проследили по референци, настао би проблем. Нит *t* наставља да живи и након што се функција *bar* заврши, а са завршетком *bar*, нестаје *i*.
- Зато се овде заправо и неће десити преношење по референци, тј. *thread* конструктор прави копију.

# Преношење параметара функцији нити, наставак

- Ако заиста хоћемо по референци, морамо писати овако:

```
struct foo {  
    void operator()(const int& x);  
};
```

```
std::thread bar() {  
    int i = 6;  
    std::thread t(foo(), std::cref(i)); // ref ако немамо const  
    return t;  
}
```

- Али тада морамо бити јако пажљиви.
- Нитске функције би требало да увек примају параметре по вредности, осим у специјалним случајевима када јако јасно знамо шта радимо.

# Преношење параметара функцији нити, наставак

- Ова правила важе и за ламбда функције:

```
std::thread t([](int i){...}, i);
```

```
std::thread t([](int& i){...}, std::ref(i));
```

- Али не важе при захватању атрибута.

```
std::thread t([i]() {...});
```

```
std::thread t([&i]() {...});
```

- У другом случају ће *i* бити стварно захваћено по референци.

# Додатне корисне ствари

- `std::thread` има још две методе које су некада корисне:

```
std::thread t(foo());  
...  
t.get_id(); // враћа јединствени идентификатор нити  
...  
t.joinable(); // проверава да ли је нит прикључива
```

- Неколико корисних ствари које се зову из саме нити:

```
void foo() {  
    std::this_thread::get_id();  
    ...  
    std::this_thread::sleep_for(  
        const chrono::duration<Rep, Period>& x);  
    // пример употребе: std::this_thread::sleep_for(2s);  
    ...  
    std::this_thread::yield();  
}
```

# Међусобна искључивост

- За конкурентно (и паралелно) програмирање карактеристичан је проблем критичне секције.
- Критична секција је део кода нити који приступа дељеном ресурсу (коме нека друга нит такође може да приступи).
- Најбоље решење проблема критичне секције јесте да се код преправи тако да не приступа дељеној променљивој.
- Друго решење је да се приступ дељеној променљивој заштити објектом искључивог приступа (мутексом).

```
std::mutex m;  
void foo() {  
    m.lock();  
    ... // приступ дељеном ресурсу  
        // само једна нит се може овде налазити  
    m.unlock();  
}
```

# Међусобна искључивост

- Мутекс се не може се копирати, нити премештати (нема мува).
- Метода `lock` је блокирајућа: Ако је мутекс већ закључан, нит ће ту чекати док се мутекс не ослободи.
- Постоји и неблокирајућа варијанта:

```
std::mutex m;  
void foo() {  
    if (!m.try_lock()) return;  
    ... // приступ дељеном ресурсу  
        // само једна нит се може овде налазити  
    m.unlock();  
}
```



# Међусобна искључивост

- Мутекс је ресурс, који се заузима и ослобађа.
- Основни принцип RAII (Resource Acquisition Is Initialization): „власништво“ над ресурсом (објектом који нема досег) доделити некој променљивој која има досег.

```
std::mutex m;  
void foo() {  
    m.lock();  
    ... // приступ дељеном ресурсу  
        // шта ако се деси изузетак овде?  
    m.unlock(); // ово можемо лако заборавити, поготово ако је  
                // ток извршавања мало разгранатији  
}  
  
void bar() {  
    std::lock_guard<std::mutex> lock(m); // у конструктору мутекс  
                                         // се заузима  
    ... // приступ дељеном ресурсу  
} // у деструктору се мутекс ослобађа; ради и за изузетке
```

# Међусобна искључивост

- Код конкурентног програмирања, једна од опасности је међусобно блокирање процеса (енгл. „deadlock“).
- До тога може доћи уколико нити морају заузети више објеката искључивог приступа (мутекса).
- Постоје разне технике за избегавање међусобног блокирања, а у оквиру стандардне библиотеке нуди се једна техника која очекује да се сви потребни мутекси заузму одједном.

```
std::mutex m1, m2;  
void foo() {  
    std::scoped_lock<std::mutex> lock(m1, m2); // или више мутекса  
    ... // приступ дељеном ресурсу  
}
```

- Мутекси ће бити заузети на начин који гарантује да неће доћи до међусобног блокирања.

# Међусобна искључивост

- `std::scoped_lock` је уведен од Це++17.
- У Це++11/14 сличну функционалност је могуће остварити комбиновањем `std::lock()` и `std::lock_guard`.

```
std::mutex m1, m2;  
void foo() {  
    std::lock(m1, m2); // или више мутекса  
    std::lock_guard<std::mutex> lg1(m1, std::adopt_lock);  
    std::lock_guard<std::mutex> lg2(m2, std::adopt_lock);  
    ... // приступ дељеном ресурсу  
}
```

- `std::lock()` заузима два или више мутекса тако да избегава међусобно блокирање.
- Када се `std::adopt_lock` проследи конструктору класе `std::lock_guard`, неће се поново заузимати мутекс, а обезбеђујемо да се мутекс ослободи у деструктору.

# Међусобна искључивост

- Постоје још неке врсте мутекса:
- Временски ограничени мутекс (`std::timed_mutex`)
  - Пружа функције за временски ограничено неблокирајуће закључавање (`try_lock_for` и `try_lock_until`)
- Рекурзивни мутекс (`std::recursive_mutex`)
  - Могуће га је рекурзивно закључавати из исте нити.
  - Због тога је процес закључавања/откључавања скупљи, па користи ово само када је потребно.
  - Постоји и временски ограничени рекурзивни мутекс
- Дељени мутекс (`std::shared_mutex`)
  - Омогућава закључавање мутекса са саопштавањем намере.
  - На тај начин могуће је да више нити приступа једном дељеном ресурсу: једна да пише, а више њих да чита.
  - Постоји и временски ограничени дељени мутекс.

# Међусобна искључивост

- Интересантан је још и класа `unique_lock`.
- То је најфлексибилнија брава.
- Омогућава временски ограничено неблокирајуће закључавање (уколико га и мутекс подржава)
- Омогућава рекурзивно закључавање (уколико га и мутекс подржава)
- Подржава премештање (мув), али не и копирање.
- Омогућава закључавање/откључавање и мимо конструктора/деструктора.

# Међусобна искључивост

- Ова могућност је врло важна

```
std::mutex m;  
void foo() {  
    std::unique_lock<std::mutex> lock(m); //  
    ... // приступ дељеном ресурсу  
    lock.unlock();  
    ... // сада је мутекс (привремено) ослобођен  
    lock.lock();  
    ... // а сада је опет заузет  
} // ослобађање мутекса у деструктору
```

# Условне променљиве

- Често имамо потребу за оваквим кодом:

```
std::mutex m;  
void foo() {  
    std::unique_lock<std::mutex> lock(m);  
    ...  
    while (!some_condition);  
    ... // ради нешто за шта је био потребан услов  
}
```

- Јасно је да се у `while` петљи губи време.
- Зато је ово боља варијанта

```
std::mutex m;  
std::condition_variable cv;  
void foo() {  
    std::unique_lock<std::mutex> lock(m);  
    ...  
    cv.wait(lock, [](){ return some_condition; });  
    ... // ради нешто за шта је био потребан услов  
}
```

# Условне променљиве

- `wait` методи се прослеђује и брава, јер док нит чека – треба да откључа браву, а чим се услов задовољи па жели да настави са радом – треба опет да је закључа. (Зато мора да се користи `unique_lock`)
- Метода је блокирајућа и зато се не губи време које иде на извршавање `while` петље.
- Међутим, када нит треба да се пробуди и поново провери услов?

```
std::mutex m;  
std::condition_variable cv;  
void foo() {  
    std::unique_lock<std::mutex> lock(m);  
    ...  
    cv.wait(lock, [](){ return some_condition; });  
    ... // ради нешто за шта је био потребан услов  
}
```



# Условне променљиве

- Па онда када јој јави нека друга нит, која можда баш промени нешто што утиче на услов.

```
std::mutex m;
std::condition_variable cv;
void foo() {
    std::unique_lock<std::mutex> lock(m);
    ...
    cv.wait(lock, [](){ return some_condition; });
    ... // ради нешто за шта је био потребан услов
}

void bar() {
    ... // нешто што утиче на some_condition
    cv.notify_one(); // или notify_all()
}
```

# Семафори

- Семафори су још један згодан начин синхронизације, који није део Це++ стандардне библиотеке, али се лако прави на основу условне променљиве.
- Формално: семафор је ненагативан цео број  $S$ , над којим су дефинисане две недељиве операције:  $V$  (signal) и  $P$  (wait).
- $V(S)$  увећава  $S$  за један.
- $P(S)$  умањује  $S$  за један, уколико  $S$  није 0.
- За семафор кажемо да је у сигнализираним стању (тј. да је сигнализирани), ако је  $S$  различито од 0. У супротном је у несигнализираним стању (несигнализирани је) и на њему се мора чекати.

# Семафори

```
struct Semaphore {
    Semaphore() = default;
    Semaphore(int x) : m_s(x) {}

    void signal() {
        std::unique_lock<std::mutex> lock(m_mut);
        m_s += 1;
        m_cv.notify_one();
    }

    void wait() {
        std::unique_lock<std::mutex> lock(m_mut);
        m_cv.wait(lock, [this]() { return m_s != 0; });
        m_s -= 1;
    }

private:
    int m_s = 0;
    std::mutex m_mut;
    std::condition_variable m_cv;
};
```

# Семафори

- Семафори су згодни за синхронизацију око кружних бафера

```
struct RingBuffer {  
    void write(int x);  
  
    int read();  
  
private:  
    std::array<int, 10> m_buff;  
    int m_w = 0;  
    int m_r = 0;  
    Semaphore m_empty{10};  
    Semaphore m_full;  
    std::mutex m_mut;  
};
```

# Семафори

- Семафори су згодни за синхронизацију око кружних бафера

```
void RingBuffer::write(int x) {  
    m_empty.wait();  
    std::lock_guard<std::mutex> l(m_mut);  
    m_buff[m_w] = x;  
    m_w = m_w % 10 == 9 ? 0 : m_w + 1;  
    m_full.signal();  
}
```

```
int RingBuffer::read() {  
    m_full.wait();  
    std::lock_guard<std::mutex> l(m_mut);  
    int res = m_buff[m_r];  
    m_r = m_r % 10 == 9 ? 0 : m_r + 1;  
    m_empty.signal();  
    return res;  
}
```

# Још неке интересантне ствари

- Остало је још неколико интересантних тема:
- Механизам добављана повратних вредности из нити (promise и future)
- Нова врста трајности променљивих: нитска трајност. Њоме нити добијају своје приватне верзије података који су до сада могли бити статичке трајности: глобалне променљиве, статичке локалне, статички атрибути... (thread\_local)
- Изражавање недељивих (атомских) операција, које јесу или нису хардверски подржаве (<atomic>)
- Али, то истражујте сами...