Вишенитно програмирање

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 узору 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 објеката 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.

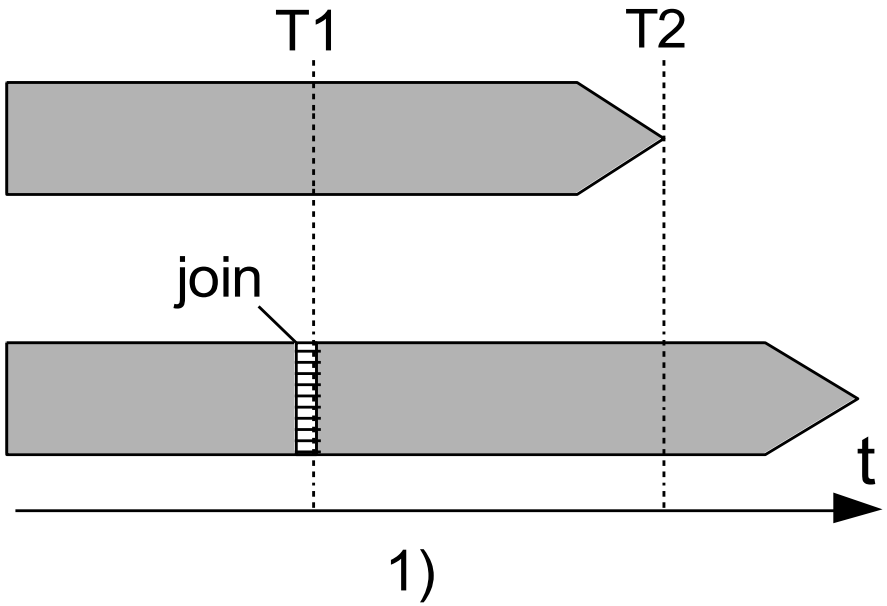
Прикључивање нити

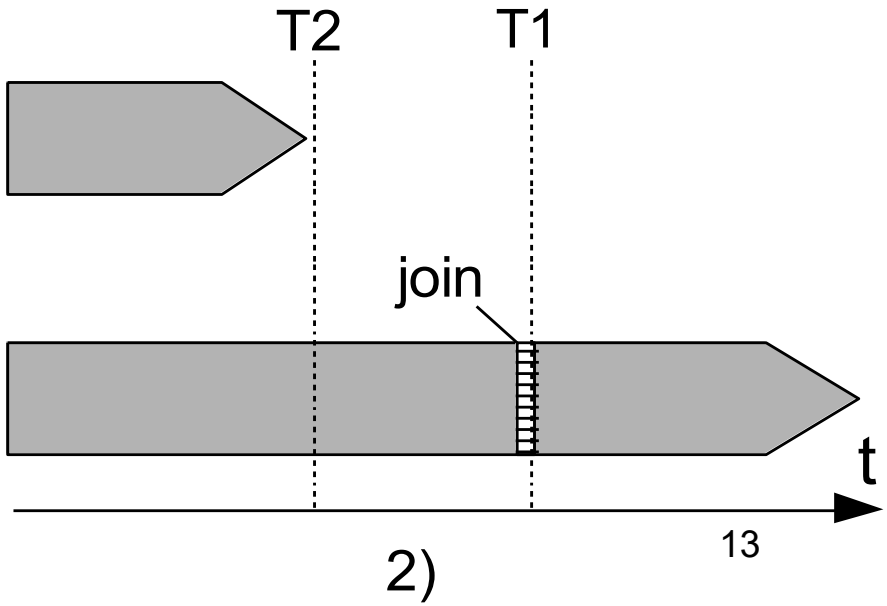
- Овакав модел синхронизације, у којем једна нит чека на завршетак друге, назива се **прикључивање нит** (енгл. join)
- На овај начин, ток програма који се извршавао у нити која се чека прикључује се току програма нити која чека, и каже се да се једна нит прикључује другој.

```
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

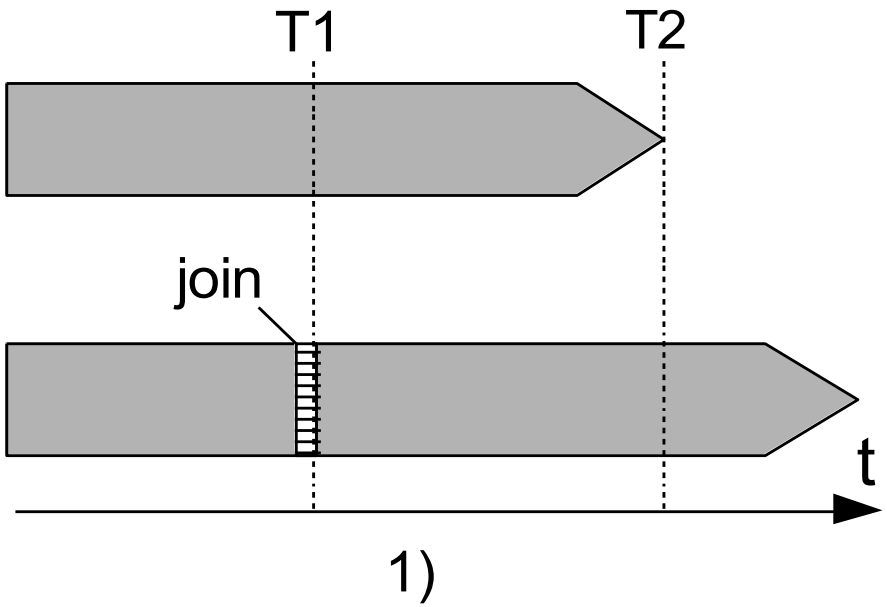
- Илустрована су два случаја чекања нити на прикључење.
- У првом случају нит којој се прикључује прелази у стање чекања (тренутак T1) пре него што је нит која се прикључује завршила свој ток (тренутак T2).
- У другом случају нит која се прикључује завршава са радом (тренутак T2) пре него што је нит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

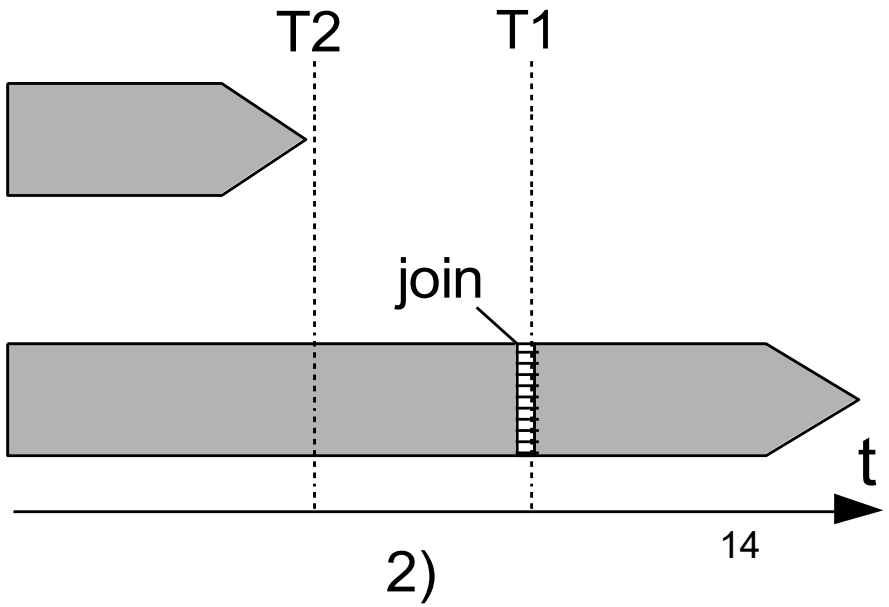




Прикључивање нити

- 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) Нову 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 сложенија дискусија.

Прикључивање нити

- 2) Нову 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 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 и 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

- Природу проблема је могуће илустровати на најједноставнијем примеру:

```
count++
```

- Ако је **count** дељени ресурс, онда се ова операције обично обавља у три корака:
  - Учитавање из меморије
  - Повећавање
  - Враћање нове вредности у меморију
- Лако је замислити шта би се могло десити ако би се ови кораци испреплетали у времену са истим тим корацима обављаним из друге нити.

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 откључа.
- 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

- 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 мутеск
- 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.
- Нит која чека ће се пробудити и проверити услов само онда када је условна променљива буде сигнализирала.

```
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 ненегативан цео број S, над којим су дефинисане две недељиве операције: V (signal/release) и P (wait/acquire).
- V(S) увећава S за један.
- P(S) умањује S за један, уколико S није 0.
- За семафор кажемо да је у сигнализираном стању (тј. да је сигнализиран), ако је S различито од 0. У супротном је у несигнализираном стању (несигнализиран је) и на њему се мора чекати.

Семафори

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

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

  void acquire() {
    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::muteks 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_free(10);
  Semaphore m_taken(0);
  std::mutex m_mut;
};
```

Семафори

- Семафори су згодни за синхронизацију око кружних бафера

```
void RingBuffer::write(int x) {
  m_free.acquire();
  {
     std::lock_guard<std::mutex> l(m_mut);
     m_buff[m_w] = x;
     m_w = m_w % 10 == 0 ? 0 : m_w + 1;
  }
  m_taken.release();
}

int RingBuffer::read() {
  m_taken.acquire();
  {
      std::lock_guard<std::mutex> l(m_mut);
      int res = m_buff[m_r];
      m_r = m_r % 10 == 0 ? 0 : m_r + 1;
  }
  m_free.release();
  return res;
}
```

Семафори у Це++20

- Семафор постаје део стандардне библиотеке.
- Две верзије:
  - counting_semaphore
  - binary_semaphore

- Спрега counting_semaphore-а је врло слична овоме што је дату у примеру на претходним слајдовима.