C++

7-8. HÉT – STD::CHRONO, SZÁLAK, SZEMAFOROK, TASKOK, C++20 COROUTINE-OK

Szálak C++-ban – szinkronizálás szemaforokkal

A C++11 óta létezik a C++ nyelvben a beépített szemafor koncepció is, amit mutex-nek nevezünk (ehhez kell a <mutex> header).

A mutex lezárásával (*lock()*) jelezzük, hogy a kinyitásáig (*unlock()*) más szál nem lockolhatja – így ha mindegyik szál azonos mutexet próbál lezárni, muszáj egymást megvárniuk.

(Gyakran) fontos, hogy a lezáráshoz tartozó szakasz gyorsan lefusson, mert a többi szál mindaddig blokkolni fog, ameddig a mutexet az azt lezáró szál fel nem oldja!

- Persze ha csak az a célunk, hogy a lockoló szál fusson végig, akkor ez nem lényeges

Szálak C++-ban – szinkronizálás szemaforokkal

```
std::mutex mtex;
auto countdown_lambda = [&mtex](int lim) {
    mtex.lock();
    for (int i = lim; i > 0; i--) { std::cout << "countdown " << i << std::endl; }
    mtex.unlock();
};

auto countup_lambda = [&mtex](int lim) {
    mtex.lock();
    for (int i = 0; i < lim; i++) { std::cout << "countup " << i << std::endl; }
    mtex.unlock();
};

std::thread thread1(countdown_lambda, 100); // a parametereket a callable utan adjuk meg
std::thread thread2(countup_lambda, 100);
thread1.join(); thread2.join();
```

Szálak C++-ban – szinkronizálás szemaforokkal

Mutexek esetében fontos az is, hogy:

- Amit szemafort lefoglaltunk, azt később ne felejtsük el elengedni (ebben segíthet az std::lock_guard, ld. később)
- Ne akarjunk véletlenül se 2x lefoglalni egy szemafort (ez nem definiált viselkedéshez vezet)
- Ne akarjunk olyan szemafort felszabadítani, amit nem birtokolunk
- Ha több szemafort foglalunk le egy feladat kontextusában, azokat az összes szálban, mindig azonos sorrendben foglaljuk le. Ha nem így teszünk, akkor előfordulhat, hogy deadlock jön létre!
  - Deadlock: Olyan szemafort akarunk lefoglalni, amit egy másik szál már lefoglalt, viszont a másik szál meg olyan szemafort akar lefoglalni, amit mi már lefoglaltunk.

Deadlock szimulációja

```
std::mutex muA, muB;

void CallHome_AB(string message) {

  muA.lock();

  //Some additional processing

  std::this_thread::sleep_for(

    std::chrono::milliseconds(100)

  );

  muB.lock();

  cout << "Thread " <<

    this_thread::get_id() <<

    " says " << message << endl;

  muB.unlock();

  muA.unlock();

}
```

```
void CallHome_BA(string message) {

  muB.lock();

  //Some additional processing

  std::this_thread::sleep_for(

    std::chrono::milliseconds(100)

  );

  muA.lock();

  cout << "Thread " <<

    this_thread::get_id() <<

    " says " << message << endl;

  muA.unlock();

  muB.unlock();

}
```

```
int main() {

thread t1(CallHome_AB,
"Hello from Jupiter");

thread t2(CallHome_BA,
"Hello from Pluto");

t1.join();

t2.join();


std::cin.get();

return 0;

}
```

Szemaforok felszabadítása helyett... (std::lock_guard)

Mivel tényleg kényelmetlen tud lenni, hogy a szemaforokat nemcsak lefoglalni kell, hanem fel is kell őket szabadítani, ezért hasznos tud lenni az ***std::lock_guard***. Ezt a template osztályt ha egy mutex átadásával példányosítjuk, lefoglalja a mutexet és a block végén automatikusan felszabadítja (amikor a destruktora meghívódik):

```
std::mutex mtex;

auto doSomething = [&mtex]() {

  std::lock_guard<std::mutex> mylock(mtex);

  std::cout << "thread has locked mutex... " << std::endl;

  std::this_thread::sleep_for(std::chrono::seconds(5));

};

std::thread myThread(doSomething);

std::this_thread::sleep_for(std::chrono::seconds(1));
```

```
{

  std::cout << "trying to acquire mylock" << std::endl;

  std::lock_guard<std::mutex> mylock(mtex);

  std::cout << "freed mylock" << std::endl;

}

std::cout << "acquiring lock AGAIN" << std::endl;

mtex.lock(); mtex.unlock();

std::cout << "joining..." << std::endl;

myThread.join();
```

Szálak C++-ban – szinkronizálás szemaforokkal

Kihasználva, hogy amikor egy szál alszik, a többi szál juthat lehetőséghez, megpróbálhatjuk kikényszeríteni, hogy egy adott szál meddig fusson (a gyakorlatban ez nem fog működni):

```
std::mutex mtex;

auto countdown_lambda = [&mtex](int lim) {// mindig 5 sort irhat ki

  mtex.lock();

  int maxPrints = 5; int numPrints = 0;

  while (true) {

    std::cout << "countdown" << lim-- << std::endl;

    if (lim == 0) { mtex.unlock(); break; }

    numPrints++;

    if (numPrints == maxPrints) {

      mtex.unlock();

      std::this_thread::sleep_for(std::chrono::milliseconds(1));

      mtex.lock();

      numPrints = 0;

    }

  }

};
```

Szálak C++-ban – szinkronizálás szemaforokkal

Kihasználva, hogy amikor egy szál alszik, a többi szál juthat lehetőséghez, megpróbálhatjuk kikényszeríteni, hogy adott szál meddig fusson (a gyakorlatban ez nem fog működni):

```
// mindig 3 sort irhat ki
auto countup_lambda = [&mtex](int lim) {
  mtex.lock(); int start = 1;
  int maxPrints = 3; int numPrints = 0;
  while (true) {
    std::cout << "countup" << start++ << std::endl;
    if (start == lim + 1) {mtex.unlock(); break; }
    numPrints++;
if (numPrints == maxPrints) { mtex.unlock();
      std::this_thread::sleep_for(std::chrono::milliseconds(1));
      mtex.lock(); numPrints = 0;
    }
  }
};   std::thread thread1(countdown_lambda, 100);
     // a parametereket a callable utan adjuk meg
    std::thread thread2(countup_lambda, 100);
    thread1.join(); thread2.join();
```

Szálak C++-ban – szinkronizálás szemaforokkal

A gyakorlatban ez nem ilyen egyszerű, mert a gépen más programok is futhatnak. Simán lehet, hogy mire a 2. szál futtatható lenne, addigra az 1. is felébred és a futtatási környezet megint azt választja ki futásra.

Ennek a problémának a kezelésére használhatunk valamilyen referenciaként átadott változót, melynek értéke meghatározhatja, hogy melyik szál futhat legközelebb

- Ez a megoldás kicsit “házilag eszkábált” és azt feltételezi, hogy a szálak, amikor felébrednek, állandóan ennek a változónak az értékét ellenőrzik, majd legtöbbször újra elalszanak… ami nem túl hatékony dolog

Ennél pár oktávval sztenderdebb / hatékonyabb megoldás az ***std::condition_variable*** osztály használata – melyre hamarosan visszatérünk, miután az ***std::unique_lock*** osztály használatával megismerkedtünk.

Szálak C++-ban – std::unique_lock

Az ***std::unique_lock*** osztály egy wrapper – többek között az ***std::mutex*** típust tudja be-wrappelni.

Ennek az osztálynak több értelme is van, pl.:

- Segítségével késleltethető a szemafor lefoglalása
- Segítségével időkorlátot lehet meghatározni a szemafor lefoglalására
- Segítségével tudjuk használni az imént hivatkozott std::condition_variable osztályt
- Destruktorában automatice elengedi a lefoglalt szemafort (amennyiben lefoglalta) – hasonlóan, mint a korábban vett ***std::lock_guard***

Nézzük a korábbi szinkronizálási feladatot, de most a unique_lock és std::condition_variable segítségével ténylegesen megoldva. Ennek a példának a megértéséhez pár dolgot tudnunk kell.

Szálak C++-ban – std::condition_variable

Az ***std::condition_variable*** osztálynak van egy wait() metódusa, amely minimálisan egy std::unique_lock objektumot vár argumentumként, de átadható neki emellett egy predikátum (igaz-hamis kimenetű) függvény is.

- A wait() használatához fontos, hogy előtte lefoglaljuk az argumentumként átadott std::unique_lock objektumot
- A wait() metódus ezt a unique_lock objektumot elengedi, és addig vár, ameddig egy másik thread meg nem hívja ugyanerre az ***std::condition_variable*** objektumra a *notify_one()* vagy *notify_all()* metódust.
- A wait() metódusnak opcionálisan át lehet adni egy második argumentumként egy predikátum-függvényt is (igaz-hamis kimenetű fv-t), ami azt fogja eredményezni, hogy addig nem megy tovább a szál futása, ameddig úgy nem történik meg a notify_one() vagy notify_all() hívás, hogy közben a predikátum függvény is igazat ad.
  - Ennek segítségével elérhető, hogy a szál mégse futhasson tovább, ha időközben egy másik szál megváltoztatta a futásához szükséges feltételeket.
- Mihelyt a wait() metóduson továbblép a program, az argumentumként átadott std::unique_lock lefoglalásra kerül.

Szálak C++-ban – std::unique_lock

Ezek alapján a megoldás:

```
std::mutex mtex;
std::condition_variable data_cond;
bool countDownIsFinished = false;
auto countdown_lambda = [&mtex, &data_cond, &countDownIsFinished](int lim) {
    int maxPrints = 5; int numPrints = 0;
    while (true) {
        std::cout << "countdown" << lim-- << std::endl;
        if (lim == 0) {
            countDownIsFinished = true; data_cond.notify_one(); break;
        }
        numPrints++;
        if (numPrints == maxPrints) {
            {
                std::unique_lock<std::mutex> mylock(mtex);
                data_cond.notify_one();
                data_cond.wait(mylock);
                // az elozo sor felszabaditja a mutexet es var a kovetkezo notify-ra
            }
            numPrints = 0;
        }
    }
}; // countdown_lambda vege
```

Szálak C++-ban – std::unique_lock

Ezek alapján a megoldás:

```
auto countup_lambda = [&mtex, &data_cond, &countDownIsFinished](int lim) {
    int start = 0; int maxPrints = 3; int numPrints = 0;
    {
        std::unique_lock<std::mutex> mylock(mtex);
        data_cond.wait(mylock); //felszabaditja mylock-ot es var a kovetkezo notify-ra
    }

     while (true) {
         std::cout << "countup" << start++ << std::endl;
         if (start == lim + 1) { break; }
         numPrints++;
         if (numPrints == maxPrints && !countDownIsFinished) {
            {
                std::unique_lock<std::mutex> mylock(mtex);
                data_cond.notify_one();
                data_cond.wait(mylock); // felszabaditja es var a kovetkezo notify-ra
                 }
                 numPrints = 0;
             }
        }
    }; // countup_lambda vege
    std::thread thread1(countdown_lambda, 100); std::thread thread2(countup_lambda, 100);
    thread1.join(); thread2.join();
```

Szálak C++-ban – szinkronizálás szemaforokkal – std::recursive_mutex

Vannak esetek, amikor szeretnénk, ha egy mutex-et többször le lehetne zárni (egyazon szálnak), és mindaddig zárva maradna, amíg ugyanannyiszor ki nem nyitottuk (többszintes, vagy rekurzív mutexek).

- Erre való az ***std::recursive_mutex*** osztály.

Ennek használata például akkor indokolt, amikor több függvény ugyanazt a mutexet foglalja le, de az egyikből a másikat is meg szeretnénk hívni. Ld. példa a következő fólián.

- Ha nem szeretnénk a lefoglalások számával bíbelődni, használhatunk egy wrappert, mint az ***std::lock_guard***!

Szálak C++-ban – szinkronizálás szemaforokkal – std::recursive_mutex

```
struct Complex {
  std::recursive_mutex mutex;
  double i;
  Complex() : i(1) {}
  void mul(int x) {
    std::lock_guard<std::recursive_mutex> lock(mutex);
    i *= x;
  }
  void div(int x) {
    std::lock_guard<std::recursive_mutex> lock(mutex);
    i /= x;
  }
  void both(int x, int y) {
    std::lock_guard<std::recursive_mutex> lock(mutex);
    mul(x);
    div(y);
  }
};

int main() {
  Complex complex;
  complex.both(32, 23);
  std::cout << complex.i;
```

Szálbiztonság std::atomic segítségével

Gyakran megkerülhetjük a mutexek használatát az ***std::atomic*** osztály használatával.

Kis adattípusok esetén sokkal gyorsabb, ha lock-free implementációt használunk – ilyet valósít meg az ***std::atomic***.

Ez az <atomic> headerben levő template-elt típus egyfajta wrappert valósít meg a template típus köré, aminek minden írása atomi.

Ha nagyobb osztályokkal használjuk, akkor gyakorlatilag megfelel a mutex-használatnak (bár szintaktikailag könnyebben használható). Egy egyszerű példa:

```
class A {
    std::atomic<int> x;
public:
    A() : x{0} {}
    void add() { x++; }
    void sub() { x--; }
};
```

Szálak helyett taskok C++-ban

Vannak esetek, amikor szálakat készíteni túlzott overheaddel jár. Ilyenkor hasznos az ***std::async()*** függvény, amely a <future> headerben található.

- Például bosszantó lehet, hogy egy már létező függvényből másik változatot kell készíteni csak azért, mert a thread-ben futó függvénynek nem lehet visszatérési típusa
- Másrészt az is nehézkes, amikor a thread-ek által kiszámított eredményt egy másik kódrésznek kell processzálnia – ilyenkor át kell adni a thread-eket (hogy bevárhassa őket), plusz referenciákat a thread-ek által manipulált változókhoz

Az ***std::async()*** egy template-elt függvény, amely template-ként vár egy függvény típust és akárhány paraméter típust – és egy adott függvény / paraméter kombó végrehajtásához tartozó std::future objektumot ad vissza

- Magyarán: az std::asnyc függvény egy std::future objektumba csomagolja az eredményt és azt adja vissza.
- Miután meghívtuk az std::async függvényt, elég a visszakapott future-t továbbítani más kódrészeknek, mivel ennek a get() metódusát bárki meghívhatja
- Első argumentumként megadhatjuk a végrehajtás “policy”-ját is – ezzel eldönthetjük, hogy a függvény külön szálon kerül végrehajtásra, vagy cask akkor, amikor az eredményhez megpróbálunk hozzáférni.
  - Ha a policy **std::launch::async**, a task külön szálon fut le; ha pedig **std::launch::deferred**, akkor a task akkor fog lefutni, amikor a visszakapott future-re meghívjuk a get() metódust

Szálak helyett taskok C++-ban

Ha az ***std::async()*** első argumentumaként nem adjuk meg, hogy async vagy deferred legyen a policy, akkor a fordítóra van bízva, hogy melyik opciót választja.

Az ***std::async()*** magasabb absztrakciós szinten működik, mint a thread-ek – ezért használata egyszerűbb, viszont kevésbé rugalmas.

- Sokszor az egyszerűbb használat átláthatóbb kódhoz vezet, ami igen nagy pozitívum
- Cserébe elveszítünk néhány lehetőséget a finomhangolásra – pl. nem tudjuk megmondani, hány szál indulhat egyszerre, hol várjuk be őket és így tovább

Nézzünk tehát néhány példát a két módszer összevetésére!

Szálak helyett taskok C++-ban

```
int temp;
//launch::deferred egyazon id-ju szalon futtatja az aszinkron muveleteket
auto future1 = std::async(
  std::launch::deferred,
  [&](int secs_to_sleep) {
    std::cout << "Thread id of future 1 is: " << std::this_thread::get_id() << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(secs_to_sleep));
    temp = 10;
  },
  10 // sleep 10 seconds
);
```

Szálak helyett taskok C++-ban

```
auto future2 = std::async(
  std::launch::deferred,
  [&]() {
    std::cout << "Thread id of future 2 is: " << std::this_thread::get_id() << std::endl;
    while (temp < 15) {
      std::this_thread::sleep_for(std::chrono::milliseconds(1));
      std::cout << ++temp << std::endl;
    }
});
future1.get();
future2.get();
```

Szálak helyett taskok C++-ban

Most nézzük meg mi a különbség a két eset között, amikor szálakat, illetve taskokat kell bevárni. Először a szálas eset:

```
void accumulate_block_worker(int* data, int count, int* result) {
  *result = std::accumulate(data, data + count, 0);
}
// Ket szalat inditunk el es a hivo feladata ezeket bevarni:
std::vector<std::thread> launch_split_workers_with_std_thread(std::vector<int>& v, std::vector<int>* results) {
  std::vector<std::thread> threads;
  threads.emplace_back(accumulate_block_worker, v.data(), v.size() / 2, &((*results)[0]));
  threads.emplace_back(accumulate_block_worker, v.data() + v.size() / 2, v.size() / 2, &((*results)[1]));
  return threads;
}
```

Szálak helyett taskok C++-ban

A workerek így indíthatóak el / várhatóak be:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8 };

std::vector<int> results(2, 0);

std::vector<std::thread> threads = launch_split_workers_with_std_thread(v, &results);

for (auto& t : threads) {

  t.join();

}

std::cout << "results: “ << results[0] << " and " << results[1] << "\n";
```

Szálak helyett taskok C++-ban

Most nézzük meg ugyanezt taskokkal. Először is a block worker fv sokkal természetesebb. Másrészt a future ök önmagukban reprezentálják az eredményt (nem kell külön explicite a thread-eket visszavárni / bevárni)

```
int accumulate_block_worker_ret(int* data, int count) {
  return std::accumulate(data, data + count, 0);
}


using int_futures = std::vector<std::future<int>>;
int_futures launch_split_workers_with_std_async(std::vector<int>& v) {
  int_futures futures;
  futures.push_back(std::async(std::launch::async, accumulate_block_worker_ret, v.data(), v.size() / 2));
  futures.push_back(std::async(std::launch::async, accumulate_block_worker_ret, v.data() + v.size() / 2, v.size() / 2));
  return futures;
}
```

Szálak helyett taskok C++-ban

Az eredmények sokkal egyszerűbben begyűjthetők:

```
// sokkal egyszerubb

std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8 };

int_futures futures = launch_split_workers_with_std_async(v);

std::cout << "results: " << futures[0].get() << " and " << futures[1].get() << "\n";
```

Szálak helyett taskok C++-ban

Amit még érdemes megemlíteni, hogy taskok / future-ök esetén a kivételkezelés is sokkal természetesebb

- (ha egy threadben futó kód miatt kivétel jön létre, azt nem lehet elkapni – legalábbis a thread-et létrehozó kódrészben, -- mivel az a kódrész egy másik szálban fut)
- std::async esetében viszont működik a catch blokk!

```
int accumulate_block_worker_ret_w_exception(int* data, size_t count) {
  throw std::runtime_error("something broke");
  return std::accumulate(data, data + count, 0);
}

// ...
```

Szálak helyett taskok C++-ban

Ez a kivétel belefut a catch blokkba (kivételeket részleteiben még nem vettünk, erre a jövőben még sor kerül)

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8 };
try {
  std::future<int> fut = std::async(std::launch::async, accumulate_block_worker_ret_w_exception, v.data(), v.size());
  std::cout << "use_worker_in_std_async computed " << fut.get() << "\n";
}
catch (const std::runtime_error& error) {
  std::cout << "caught an error: " << error.what() << "\n"; // el is kapja ahogy kell
}
```

Röviden a C++-20 néhány újításáról

A C++-20 újításai mai szemmel zavarba ejtően sokrétűek.

Persze még évekbe fog telni, mire ezek széleskörben elterjednek, azonban több dolgot rendesen meg is fognak változtatni.

Hogy cask egy példát említsünk, a szabványban szerepel egy ***std::jthread*** nevű típus, ami olyan, mint egy ***std::thread***, csak automatikusan join-olódik a fő szálhoz, és még kívülről is megszakítható

- Következésképpen több olyan korlát, ami a szálakkal kapcsolatban a mai órán felmerült, a jthread-ekre nem igazán érvényes

Röviden a C++-20 néhány újításáról

Egy további érdekesség a C++-20 szabványtól (most a teljesség igénye nélkül) a *coroutine*-ok.

A coroutine lényegében egy olyan függvény, melynek:

- Definíciója return utasítás helyett egy vagy több co_await, co_yield és /vagy co_return utasítást tartalmaz (ettől lesz az adott függvény coroutine)

Ha egy függvény a fenti követelményeknek megfelel, coroutine lesz az alábbi tulajdonságokkal:

- Futása a stack-től független, és csak a heap-en allokált memóriától függ – ez teszi lehetővé, hogy futás közben meg lehessen / meg tudja magát szakítani (a belső változói megmaradnak, mert nem a stack-en kapnak helyet!). A heapen foglalnak helyet tehát az (alapesetben érték szerint lemásolt) argumentumok és a belső változók. Referenciát is át lehet adni, de az eredeti objektumnak élnie kell végig, amíg a coroutine fut!
- Tartozik hozzá egy promise objektum, melyet önmagán belül manipulálhat, és amely szintén a heap-en foglal helyet
- Végezetül minden coroutine-t egy coroutine handle útján lehet kívülről manipulálni (meg lehet szakítani, destruálni lehet)

Röviden a C++-20 néhány újításáról

Amikor egy coroutine-t meghívunk, new operátorral lefoglalja a heapen a szükséges memóriát, elkezd futni. Ha co_return utasításhoz ér, a coroutine futása véget ér és a memóriája megszűnik.

Amikor egy coroutine-on belül meghívjuk a co_yield utasítást valamilyen értékkel, akkor a coroutine-t meghívó függvény visszakapja az adott értéket és a coroutine futása felfüggesztett állapotba kerül. Általában ez a visszaadott típus olyan, hogy segítségével később folytatható a coroutine futása (pl. egy generator, amin valamilyen feltétel teljesüléséig iterálni lehet

```
#include <cppcoro/generator.hpp>

cppcoro::generator<std::string> produce_items()
{
  while (true)
  {
     auto v = rand();
     using namespace std::string_literals;
     auto i = "item "s + std::to_string(v);
     print_time();
     std::cout << "produced " << i << '\n';
     co_yield i;
  }
}
```

```
cppcoro::task<> consume_items(int const n)
{
  int i = 1;
  for(auto const& s : produce_items())
  {
     print_time();
     std::cout << "consumed " << s << '\n';
     if (++i > n) break;
  }

  co_return;
}
```

Röviden a C++-20 néhány újításáról

A co_await működését szabályokkal elég bonyolult leírni (le lehet, de kicsit túlmutat a tárgy jelenlegi keretein)

Amikor egy coroutine-on belül meghívjuk a co_await utasítást, pl. valamilyen függvényhívással, akkor a coroutine felfüggesztett állapotba kerül és az adott függvény kezd el futni. A függvény visszatérésként egy olyan objektumot kell hogy adjon, melynek bizonyos operátorai definiáltak:

- bool await_ready()
- void await_suspend(std::coroutine_handle h)
- void await_resume()

Ezek tulajdonképpen callback-ek amik különböző eseményekre kerülnek meghívásra. Az ilyen objektumokat awaitable-nek nevezzük, és egyébként indirekt módon is létre tud hozni ilyet a futtatási környezet (tehát nem feltétlenül egy awaitable-t visszadó függvényt kell megadni, hanem bármit, ami ilyen awaitable-re konvertálható)

Röviden a C++-20 néhány újításáról

Álljon itt egy vázlatos példa a cppreference.com alapján. Valahogy úgy fogható fel itt a működés, hogy:

- Elindítjuk a resuming_on_new_thread() függvényt, ami co_awaittel meghív egy awaitable-t visszaadó függvényt
- Az awaitable objektumnak meghívódik az await_ready() függvénye, de ez rögtön false-t ad vissza, ezért…
- Az awaitable objektumnak meghívódik az await_suspend() függvénye. Ez megkapja az eredeti coroutine-ra hivatkozó handle-t, melynek külön szálon meghívja a resume() metódusát. Ezzel elértük, hogy az eredeti coroutine egy másik szálon folytassa futását!

```
#include <coroutine>
#include <iostream>
#include <stdexcept>
#include <thread>

auto switch_to_new_thread(std::jthread& out) {
  struct awaitable {
    std::jthread* p_out;
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
      std::jthread& out = *p_out;
      if (out.joinable())
        throw std::runtime_error("Output jthread parameter not empty");
      out = std::jthread([h] { h.resume(); });
      // Potential undefined behavior: accessing potentially destroyed *this
      // std::cout << "New thread ID: " << p_out->get_id() << '\n';
      std::cout << "New thread ID: " << out.get_id() << '\n'; // this is OK
    }
    void await_resume() {}
  };
  return awaitable{&out};
}
```

```
task resuming_on_new_thread(std::jthread& out) {
  std::cout << "Coroutine started on thread: " << std::this_thread::get_id() << '\n';
  co_await switch_to_new_thread(out);
  // awaiter destroyed here
  std::cout << "Coroutine resumed on thread: " << std::this_thread::get_id() << '\n';
}

int main() {
  std::jthread out;
  resuming_on_new_thread(out);
}
```