Threadkonzept von C++

Seminar - Nebenläufige Programmierung von Multicore-Systemen

Fin Bießler

TUHH

April 24, 2021

Einführung

- **Motivation:** Separation of Concerns und Performance
- **"Free lunch is over"** (Herb Sutter, 2005)
- Standardisierter Support für Multithreading für C++ seit C++11 (Standard C++ Thread Library)

Agenda

Grundlegenede Thread-Verwaltung

Grundlegende Thread-Verwaltung - Überblick

- Jedes C++ Programm hat mindestens einen Thread
- Dieses Programm kann jedoch weitere Threads starten
- Ein Thread wird beendet und verlassen wenn die mit ihm assozierte Funktion beendet wird

Grundlegende Thread-Verwaltung - Starten eines Threads

- Ein Thread wird immer gestartet durch das Konstruieren eines `std::thread` Objektes

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

- Die `<thread>` Bibliothek muss eingebunden werden

Grundlegende Thread-Verwaltung - Starten eines Threads

- Das Starten eines Threads funktioniert mit jedem Callable-Type

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

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

- Übergebenes Funktions-Objekt wird in den lokalen Speicher des Threads kopiert und dort ausgeführt

Grundlegende Thread-Verwaltung - Starten eines Threads

- Beim starten eines Threads kann es zu syntaktischen Überschneidungen kommen
- Kann gleich einer Funktionsdeklaration sein
- Daher gibt es die *uniform initialization syntax*
- Lambda-Expressions können ebenfalls übergeben werden

```
std::thread my_thread(background_task()); // function decl.
std::thread my_thread((background_task()));
std::thread my_thread{background_task()};
```

Grundlegende Thread-Verwaltung - Resultierende Probleme

- Es muss **explizit** angegeben werden ob auf die Beendigung eines Threads gewartet werden soll (joinen) oder ob er im Hintergrund weiterlaufen soll (detachen)
- Wenn diese Angabe nicht gemacht wird **bevor** das Thread-Objekt zerstört wird wird das programm durch den Aufruf von `std::terminate` terminiert

Grundlegende Thread-Verwaltung - Resultierende Probleme

```
struct func {
    int& i;
    func(int& i_):i(i_){}
    void operator()()
    {
        for(unsigned j=0;j<1000000;++j)
        {
            do_something(i);
        }
    }
}

void oops() {
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
}
```

Grundlegende Thread-Verwaltung - Resultierende Probleme

- **LÖSUNG:**
  - Daten in den Thread kopieren anstelle sie zu teilen (Call-By-Value anstelle von Call-By-Reference)
  - Mit dem Thread joinen

```
struct func {
    int i;
    func(int i_):i(i_){}
    ...
```

Grundlegende Thread-Verwaltung - Mit Threads joinen

- Um auf die Beendigung eines Threads zu warten muss nur `.join()` des mit ihm assoziierten Thread-Objektes aufgerufen werden

---

```
...
  void oops() {
      int some_local_state=0;
      func my_func(some_local_state);
      std::thread my_thread(my_func);
      my_thread.join();
  }
```

Grundlegende Thread-Verwaltung - Mit Threads joinen

- Fortschrittlichere Methoden zum das warten auf Threads existieren, siehe **Condition Variables** oder **Futures**
- Man Kann einen Thread nur einmal joinen
  - Wird ein Thread gejoint returnt der Aufruf von `joinable()` auf diesen Thread künftig `false`
- Ein Thread kann normalerweise sofort detached werden, wenn gewollt
- Sollte ein Thread gejoint werden muss der Entwickler gut überlegen wann um undefiniertes Verhalten zu vermeiden

Grundlegende Thread-Verwaltung - Threads detachen

- Ein Thread wird detached indem die Methode `detach()` des Thread-Objektes aufruft
- Mit einem detachtem Thread kann nicht mehr direkt kommuniziert werden
- Diese Threads werden auch oft **deamon threads** genannt

Parameterübergabe an Thread-Funktionen

Parameterübergabe an Thread-Funktionen - Überblick

- Parameter werden als extra Parameter des Threadkonstruktors an die Thread-Funktion übergeben
- **Default:** Argumente werden als rvalues in den internen Speicher des Threads kopiert.

```
void f(int i,std::string const& s);
std::thread t(f,3,''hello'');
```

Parameterübergabe an Thread-Funktionen

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

Parameterübergabe an Thread-Funktionen

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

Parameterübergabe wenn diese nur gemoved werden können

- `std::unique_ptr` könnte beispielsweise nur gemoved werden
- Eine Parameterübergabe würde daher wie folgt aussehen:

```
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));
```

- Note: `std::thread` hat das selbe Besitzverhalten wie `std::unique_ptr`

Besitz eines Threads übertragen

Besitz eines Threads übertragen - Motivation & Überlick

- Funktion die einen Thread startet dann aber den Besitz an die aufrufende Funktion zurückgibt
- Einen Thread erstellen dessen Besitz in eine andere Funktion hineingegeben werden soll

```
void some_function();
void some_other_function();
std::thread t1(some_function);
std::thread t2=std::move(t1);
t1=std::thread(some_other_function);
std::thread t3;
t3=std::move(t2);
t1=std::move(t3);
```

Besitz eines Threads übertragen

- Besitz eines Threads aus einer Funktion geben:

```
std::thread f()
{
    void some_function();
    return std::thread(some_function);
}
std::thread g()
{
    void some_other_function(int);
    std::thread t(some_other_function,42);
    return t;
}
```

Besitz eines Threads übertragen

- Besitz eines Threads in eine Funktion geben:

```
void f(std::thread t);
void g()
{
    void some_function();
    f(std::thread(some_function));
    std::thread t(some_function);
    f(std::move(t));
}
```

Automatisiertes Thread-Management

- Thread-Objekt kann in Containern die move-aware sind gespeichert werden wie `std::vector<>`
- Wie hier:

```
std::vector<std::thread> threads;
for(unsigned i=0;i<20;++i)
{
    threads.emplace_back(do_work,i);
}
for(auto& entry: threads)
    entry.join();
```

Auswahl der Anzahl der Thread-Anzahl zur Laufzeit

Auswahl der Anzahl der Thread-Anzahl zur Laufzeit - Ausblick

- Die C++ Standard Library stellt die Funktion `std::thread::hardware_concurrency` bereit
  - Anzahl der Threads die gleichzeitig auf einem System laufen können
- Kann genutzt werden um die Anzahl der Threads zur Laufzeit festzulegen

Identifizierung von Threads

Auswahl der Anzahl der Thread-Anzahl zur Laufzeit - Ausblick

- Threads haben eine Id vom Typ `std::thread::id`
  - Aufruf von `get_id()` von einem Thread-Objekt
  - Aufruf von`std::thread::get_id()`
- Können kopiert und verglichen werden

Zusammenfassung und Ausblick

Zusammenfassung und Ausblick

- Threads werden erzeugt durch das erzeugen eines `std::thread` Objektes mit der übergabe einer Einstiegsfunktion
- Könenn gejoint oder detached werden
  - Join: Es wird auf das Ende des Threads gewartet
  - Detach: Der Thread läuft im Hintergrund weiter
- Besitz eines Threads verhält sich wie der eines `std::unique_ptr`