Advanced Software Development

Multithreading

Übersicht

- Threads
- Synchronisierungsmechanismen
  - Mutexes und Locks
  - Bedingungsvariablen
  - Atomare Variablen
- Asynchrone Berechnung von Werten

Threads (1)

- Klasse std::thread in <thread>
- Starten eines Threads durch Erzeugen eines thread-Objekts

```
void f1();
void f2(int);

struct f3 {
  void operator()(double, int);
};

thread t1(f1);
thread t2(f2, 5);
thread t3(f3(), 1.5, 1);
```

Threads (2)

- Für Funktionen mit beliebig vielen Parametern
- Trennung von Deklaration und Starten

```
thread t1;
…
t1 = thread(f1);
```

- Warten auf Beendigung eines Threads

```
t1.join();
```

Threads (3)

- thread-Objekt von Thread lösen: detach
  - Thread läuft bis zum Ende weiter
  - thread-Objekt repräsentiert keinen Thread mehr
- Destruktor verursacht Programmabbruch (std::terminate) falls Thread noch aktiv

Threads – Beispiel (1)

```
class Producer {
public:
  Producer(Buffer &b) : mBuffer(b) {}
  void run();
private:
  Buffer &mBuffer;
};

void Producer::run() {
  while (…) {
    Item *item = new Item;
    …
    mBuffer.Put(item);
  }
}
```

Threads – Beispiel (2)

```
class Consumer {
public:
  Consumer(Buffer &b):mBuffer(b) {}
  void run();
private:
  Buffer &mBuffer;
};

void Consumer::run() {
  while (…) {
    unique_ptr<Item> item(mBuffer.Get());
    // ...
  }
}
```

Threads – Beispiel (3)

```
int main() {
  Buffer buf;
  Producer producer(buf);
  Consumer consumer(buf);
  thread producerThread(mem_fn(&Producer::run), &producer);
  // oder alternativ:
  // thread producerThread(bind(&Producer::run, ref(producer)));
  // bzw. mit Lambda:
  // thread producerThread([&producer] { producer.run(); });
  thread consumerThread(mem_fn(&Consumer::run), &consumer);
  producerThread.join();
  consumerThread.join();
  return 0;
}
```

Beispiel: Primzahlen zählen

Laufzeitmessung auf System mit 2x QuadCore Xeon 2.5 GHz

Mittlere Laufzeiten aus 10 Läufen:

Nr.	Zeit	Speedup
1	0,992s	
2	0,500s	1,98
4	0,252s	3,94
8	0,128s	7,74
9	0,214s	4,64

Mutexes

- Zur Synchronisation von Code-Abschnitten
- Mutex anfordern
  - Methode lock(): Thread wartet, bis Mutex frei (d.h. kein anderer Thread das Mutex hält)
  - Methode try_lock(): sperrt Mutex nur falls frei und kehrt ansonsten sofort zurück – Rückgabewert gibt an, ob erfolgreich
- Mutex abgeben: Methode unlock()

Mutex-Typen

- mutex: Standard-Mutex
- timed_mutex: erlaubt Angabe einer maximalen Wartezeit
- recursive_mutex: Thread kann mehrmals lock() auf Mutex, das er bereits hält aufrufen
- recursive_timed_mutex

Mutex – Beispiel

```
class Buffer {
public:
  void Put(Item *item);
  Item *Get();
private:
  std::deque<Item*> mItems;
  std::mutex mMutex;
};

void Buffer::Put(Item *item) {
  mMutex.lock();
  mItems.push_back(item);
  mMutex.unlock();
}
…
```

Achtung: Diese Art der Verwendung eines Mutex ist fehleranfällig und nicht exception-safe

lock_guard

- Implementiert RAII-Technik für Mutexes
- Konstruktion aus freiem Mutex

```
lock_guard<mutex> l(someMutex);  // ruft someMutex.lock() auf
```

- Konstruktion aus bereits vom aktuellem Thread gehaltenem Mutex

```
someMutex.lock();
…
lock_guard<mutex> l(someMutex, adopt_lock);
```

- Aufruf von unlock() im Destruktor => automatische Freigabe des Mutex beim Verlassen des Blocks

lock_guard – Beispiel

```
void Buffer::Put(Item *item) {
  lock_guard<mutex> l(mMutex);
  mItems.push_back(item);
}

Item *Buffer::Get() {
  lock_guard<mutex> l(mMutex);
  auto item = mItems.front();
  mItems.pop_front();
  return item;
}
```

unique_lock

- Zusätzliche Möglichkeiten zu lock_guard
  - Aufruf von try_lock() statt lock()

```
unique_lock<mutex> l(m, try_to_lock);
if (l.owns_lock()) …  // oder einfach: if (l) bzw. if (!l)
```

  - lock() erst zu einem späteren Zeitpunkt aufrufen

```
unique_lock<mutex> l(m, defer_lock);
...
l.lock();  // auch möglich: l.try_lock()
```

  - Vorzeitiger Aufruf von unlock() möglich
- Automatischer Aufruf von unlock() im Destruktor, falls nötig

Mehrere Locks gleichzeitig

- Globale lock-Funktionen erlauben sicheres gleichzeitiges Anfordern von mehreren Locks
- Beispiel:

```
// Variante 1
lock(mutex1, mutex2);
lock_guard<mutex> l1(mutex1, adopt_lock);
lock_guard<mutex> l2(mutex2, adopt_lock);

// Variante 2
unique_lock<mutex> l1(mutex1, defer_lock);
unique_lock<mutex> l2(mutex2, defer_lock);
lock(mutex1, mutex2);
```

Bedingungsvariablen (1)

- Thread wartet, bis eine Bedingung erfüllt ist (immer in Kombination mit unique_lock)

```
condition_variable cond;
…
unique_lock<mutex> l(someMutex);
while (!ready) {
  cond.wait(l);
}
```

- Interner Ablauf von *cond.wait(l)*
  - Aufruf l.unlock()
  - Warten bis Bedingungsvariable signalisiert
  - Aufruf l.lock()

Bedingungsvariable (2)

- Signalisierung

```
{
  lock_guard l(someMutex);
  …
  ready = true;
}
cond.notify_one();
```

- Signalisierungsmethoden
  - notify_one(): Einen wartenden Thread aufwecken
  - notify_all(): Alle wartenden Threads aufwecken
- condition_variable_any: für alle Lock- und Mutex-Typen, die lock() und unlock() bieten

condition_variable – Beispiel

```
void Buffer::Put(Item *item) {
  {
    lock_guard<mutex> l(mMutex);
    mItems.push_back(item);
  }
  mCond.notify_one();
}

Item *Buffer::Get() {
  unique_lock<mutex> l(mMutex);
  while (mItems.empty()) {
    mCond.wait(l);
  }
  // Statt der Schleife kann wait auch mit einem Prädikat aufgerufen werden:
  // mCond.wait(l, [this] { return !mItems.empty(); });
  Item *item = mItems.front();
  mItems.pop_front();
  return item;
}
```

Atomare Variablen

- Bieten atomare Operationen für einfache Datentypen
- std::atomic<T>
- Operatoren: ++, --, +=, -=, &=, |=, ^=
- Beispiel: Referenzzähler

```
class RefCounted {
public:
  void incRef() { refCount++; }
  int decRef() { return --refCount; }
private:
  std::atomic<int> refCount = 1;
};
```

Asynchrone Berechnung

- std::async zur asynchronen Berechnung eines Werts

```
int ComputeSomeValue(int x);

int x = ...;
// starte asynchrone Berechnung
auto val = async(ComputeSomeValue, x);
// führe weitere Aktionen durch
int result = val.get();  // warte auf Ergebnis
```

- Rückgabewert von std::async ist ein Objekt vom Typ std::future<T>

Futures

- Future steht für einen asynchron berechneten Wert
- get() Methode wartet bis Wert verfügbar und gibt diesen (per Move-Operation) zurück
- get() darf nur einmal aufgerufen werden (außer bei std::shared_future)

std::async und Threads

- Berechnung kann, muss aber nicht in einem eigenen Thread erfolgen
- Explizite Angabe einer „Launch Policy“ als ersten Parameter von std::async

```
// Berechnung in eigenem Thread
auto f = async(launch::async, MyTask());
auto result = f.get();  // Warten bis Berechnung fertig

// Verzögerte Berechnung
auto f = async(launch::deferred, MyTask());
auto result = f.get();  // Berechnung hier
```