

Fiber Pool API

Fiber Pool Synchronization Library

Fiber Pool Algorithm Library

[Dokumentation](#)

Copyright © 2009 ThinkMeta Software UG (haftungsbeschränkt)

Version 1.0.0.8

1 Über Fiber Pool 5

1.1 Spezifikation der Fibers	5
1.2 Konzept	6
1.2.1 Asynchrone Tasks.....	6
1.2.2 Synchronisation.....	6
1.3 Architektur	6
1.3.1 Fiber Pool Task Scheduler	7
1.3.2 Fiber Pool API.....	8
1.3.3 FiberPool Synchronization Library	8
1.3.4 Fiber Pool Algorithm Library	8
1.4 Spezielle Komponenten	8
1.4.1 File I/O Scheduler.....	9
1.4.2 Verwaltung von virtuellem Speicher.....	9

2 Fiber Pool verwenden 9

2.1 Fiber Pool für ein Projekt einrichten	9
2.1.1 Die FiberPoolAPI-Bibliotheken erstellen.....	10
2.1.2 Die Projekteigenschaften konfigurieren	10
2.2 Konzeptionelle Einschränkungen	10
2.3 Erste Schritte	11
2.3.1 Einen Fiber Pool initialisieren.....	11
2.3.2 Den Scheduler starten und beenden	12
2.3.3 Einen Thread für die Kommunikation mit Tasks einrichten	12
2.3.4 Eine Task implementieren	12
2.3.5 Eine Task ausführen	13
2.4 Synchronisationsobjekte	14
2.4.1 Programmbereiche für exklusiven Zugriff sperren	14
2.4.2 Ereignisbehandlung.....	16
2.4.3 Zugriff auf Ressourcen begrenzen	17
2.4.4 Gruppen	17
2.4.5 Transitionen	17
2.4.6 Auf die Berechnung einer Variablen warten	18
2.4.7 Eigene Synchronisationsklassen entwickeln.....	18

2.5 Virtueller Speicher	22
2.6 Dateien.....	23

1 Über Fiber Pool

Fiber Pool ist ein Framework für C++-Programmierer zur Entwicklung von hochskalierbaren Anwendungen unter Microsoft® Windows® XP und Vista.

Je nach implementiertem Algorithmus skaliert Fiber Pool die Anwendung in Abhängigkeit von

- der Anzahl der Prozessoren/Kerne,
- der Anzahl der Festplatten, und
- der Größe des Hauptspeichers.

Eine klassenbasierte Schnittstelle kapselt den direkten Zugriff auf die Hardwarekomponenten.

Für die Verwendung von Fiber Pool werden durchschnittliche Kenntnisse in C++ sowie in Multithreading vorausgesetzt.

Für die Erweiterung des Frameworks werden gute Kenntnisse in Multicore-Programmierung vorausgesetzt.

Folgende Betriebssysteme werden unterstützt:

- Windows XP 32-Bit
- Windows Vista 32-Bit
- Windows Vista 64-Bit (AMD64/Intel 64)

Unter 64-Bit müssen die Prozessoren den Befehl **cmpxchg16b** unterstützen.

Der bereitgestellte Quellcode sowie zugehörige Projektdateien können mit Microsoft® Visual Studio® 2008 verwendet werden.

1.1 Spezifikation der Fibers

Fiber Pool verwendet in allen Versionen eigene Fiber-Implementierungen, die nicht kompatibel mit den vom Betriebssystem bereitgestellten Fibers sind. Eine gemischte Nutzung ist nicht möglich.

In der 32-Bit-Version haben Fibers folgende Eigenschaften:

- Die Stack-Größe ist 64 KiB.
- Die Fiber-Daten werden im Thread in Adresse FS:[16] gespeichert.
- Beim Kontextwechsel werden die Register EBX, EDI, ESI und EBP sowie der SEH-Rahmen gesichert.

64-Bit-Fibers haben folgende Eigenschaften:

- Die Stack-Größe ist 64 KiB.
- Die Fiber-Daten werden im Thread in Adresse GS:[32] gespeichert.
- Beim Kontextwechsel werden die Register RBX, RDI, RSI, RBP, R12 bis R15, XMM6 bis XMM15, der Registerstatus, das Floating Point Control Word sowie der Aktivierungskontext für SxS gesichert.

1.2 Konzept

Die Programmierung von Anwendungen mit Fiber Pool basiert auf dem Konzept, so viele Vorgänge wie möglich asynchron auszuführen und sie erst bei Bedarf sehr spät zu synchronisieren. Dadurch wird die zeilenweise Synchronisation bei der Befehlsverarbeitung gelöst, die durch das imperative Programmiermodell festgelegt ist.

Da die einzelnen Vorgänge für ein gemeinsames Ziel asynchron ausgeführt werden, werden sie von Fiber Pool mittels kooperativem Multitasking ausgeführt. Dadurch müssen sie - im Gegensatz zu präemptivem Multitasking - nicht gegeneinander um Ressourcen konkurrieren.

Um kooperatives Multitasking zu ermöglichen, werden die Vorgänge in sog. Fibers ausgeführt.

Fibers sind viel flexibler als die vom Betriebssystem bereitgestellten Threads und haben die Eigenschaft, dass sie nach einer Unterbrechung grundsätzlich auf einem beliebigen Thread fortgesetzt werden können. Diese Eigenschaft erlaubt dem Fiber Pool Framework, eine sehr gute Lastbalancierung zu erreichen.

1.2.1 Asynchrone Tasks

Für die asynchrone Ausführung von Vorgängen verwendet das Fiber Pool Framework den taskbasierten Ansatz: Jeder Vorgang, der asynchron ausgeführt werden kann, wird als Task implementiert und an den Fiber Pool Scheduler zur Bearbeitung übergeben.

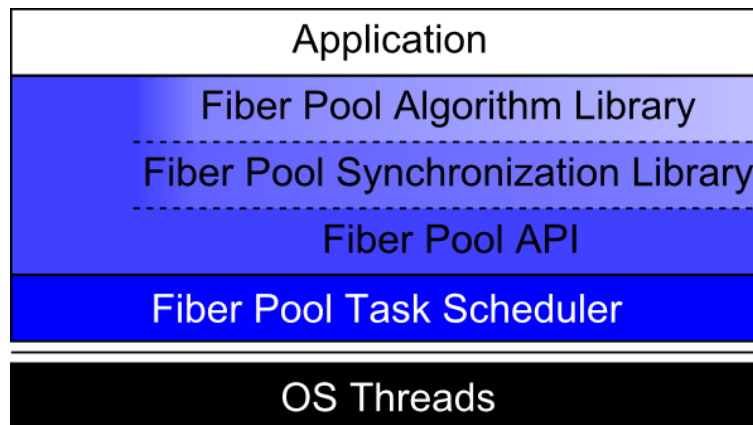
Eine Task kann während ihrer Ausführung mit anderen Tasks kommunizieren, selbst wenn diese zu diesem Zeitpunkt nicht ausgeführt werden.

1.2.2 Synchronisation

Die Kommunikation zwischen Tasks geschieht indirekt über gemeinsam genutzte Synchronisationsobjekte. Fiber Pool stellt dazu eine Vielzahl an Klassen für verschiedene Szenarien zur Verfügung.

1.3 Architektur

Das Fiber Pool Framework besteht aus vier aufeinander aufbauenden Komponenten (s. Abbildung):



Der Fiber Pool Task Scheduler ist die einzige Komponente, die lediglich als Binärdatei (FiberPool32.dll bzw. FiberPool64.dll) zur Verfügung gestellt wird.

Die anderen Komponenten werden als Quellcode innerhalb des FiberPoolAPI-Projekts zur Verfügung gestellt.

1.3.1 Fiber Pool Task Scheduler

Der Fiber Pool Task Scheduler ist für die Ausführung von Tasks zuständig. Er besteht aus einem Thread Pool sowie aus einer dynamischen Anzahl an Fibers.

Das Scheduling wird in drei Stufen durchgeführt:

In der **ersten Stufe** wird der Scheduling-Algorithmus auf die Fiber des Threads ausgeführt. Er wählt nach dem FIFO-Prinzip die nächste Task aus, die auf diesem Thread ausgeführt werden soll. Hierbei werden fortzusetzende Tasks gegenüber neuen Tasks priorisiert behandelt.

Hat der Thread in seiner lokalen Liste keine fortzusetzenden Tasks, so versucht er eine fortzusetzende Task aus einem anderen Thread des Pools auszuwählen, bevor er eine neue Task auswählt.

Enthält die lokale Liste auch keine neuen Tasks, so versucht der Thread eine neue Task aus einem anderen Thread des Pools auszuwählen.

Konnte keine Task zur Ausführung ausgewählt werden, so wird der Thread vom Algorithmus angehalten, bis neue Tasks verfügbar werden.

Handelt es sich bei der ausgewählten Task um eine fortzusetzende Task, so wechselt der Algorithmus zur Fiber, die diese Task ausführt (dritte Stufe).

Bei einer neuen Task wechselt der Algorithmus zu einer neuen Fiber und geht zur **zweiten Stufe** über:

Das Auswahlverfahren der zweiten Stufe entspricht dem der ersten Stufe.

Bei Auswahl einer fortzusetzenden Task wechselt der Algorithmus für ihre Ausführung zurück zur Fiber des Threads und zur ersten Stufe. Ein Fortsetzen einer Task ist in der zweiten Stufe nicht möglich.

Der Wechsel zur ersten Stufe erfolgt auch, wenn keine Task ausgewählt werden konnte.

Neue Tasks werden direkt auf dieser Fiber in der **dritten Stufe** ausgeführt:

Sofern keine Synchronisation mit anderen Tasks notwendig ist, wird die Task in dieser Stufe vollständig ausgeführt und der Algorithmus kehrt danach zur zweiten Stufe zurück.

Führt eine Synchronisation zum Blockieren der Task, so wechselt der Algorithmus zur Fiber des Threads und kehrt direkt zur ersten Stufe zurück.

1.3.2 Fiber Pool API

Der Zugriff auf den Fiber Pool erfolgt über das Fiber Pool API, das Basisdienste über entsprechende Schnittstellen bereitstellt.

Die Kernfunktionen sind

- das Erzeugen und Löschen von Fiber Pools,
- das Hinzufügen von Tasks, und
- das Unterbrechen und Fortsetzen von Tasks.

Zur Nutzung des Fiber Pools ist die Verwendung dieser Komponente erforderlich.

1.3.3 FiberPool Synchronization Library

Um die Verwaltung unterbrochener Tasks zu vereinfachen, stellt Fiber Pool eine Klassenbibliothek mit einer Vielzahl an Synchronisationsprimitiven bereit.

Hierzu zählen die aus der Thread-Synchronisation bekannten Primitiven wie „CriticalSection“, „Event“ und „Semaphore“ sowie die Primitiven für die Synchronisation asynchroner Abläufe wie „TaskGroup“ und „Transition“.

Für die Entwicklung eigener Synchronisationsklassen werden Basisklassen bereitgestellt, die die Taskverwaltung kapseln.

Die Verwendung der Klassenbibliothek ist optional.

1.3.4 Fiber Pool Algorithm Library

Neben den Synchronisationsprimitiven stellt die Bibliothek auch komplexe Klassen und Algorithmen bereit, die diese Primitiven einsetzen. Beispiele hierfür sind die Klassen für Dateien und virtuellen Speicher sowie der parallele Quicksort-Algorithmus.

1.4 Spezielle Komponenten

Zur Skalierung einer Anwendung hinsichtlich der Anzahl genutzter Festplatten oder der Größe des Hauptspeichers werden spezielle Komponenten angeboten:

1.4.1 File I/O Scheduler

Der File I/O Scheduler serialisiert Dateizugriffe auf einer Festplatte bzw. parallelisiert sie auf mehreren Festplatten. Dies ist für Anwendungen sinnvoll, in denen mehrere Tasks, die auf Dateien zugreifen, parallel ausgeführt werden.

In Kombination mit den verwendeten Speicherklassen kontrolliert er die Ausführung der I/O-Tasks, sodass sie bei einer Verletzung der Speicherbedingungen blockieren und erst wieder fortgesetzt werden, wenn sie sich aufgelöst hat. Dieses Verfahren erlaubt z.B. ein durchgängiges Lesen einer Datei in den Speicher bis zur seiner physischen Grenze, das dann zum Umschalten zum durchgängigen Schreiben in eine Datei führt.

Der File I/O Scheduler ist aktuell als Task implementiert und ist dem Fiber Pool Task Scheduler untergeordnet. Daher ist die Priorität, die Prozessorperformance zu steigern, höher als die Steigerung der I/O-Performance.

Unter bestimmten Umständen ist die parallele Bearbeitung von Dateien mit einer Festplatte möglich, etwa dann, wenn der Dateizugriff kürzer als die Verarbeitung ist und die verarbeitende Task nicht optimal auf alle Prozessoren skaliert (z.B. Kompressionsalgorithmen).

1.4.2 Verwaltung von virtuellem Speicher

Für die kontrollierte Verwendung von Hauptspeicher stellt die Klassenbibliothek für den sequentiellen Zugriff mehrere Speicherklassen zur Verfügung.

Aufgabe dieser Klassen ist zum einen die Reservierung, Belegung und Freigabe von virtuellem Speicher, zum anderen die regelmäßige Überprüfung des Speicherzustands, der ggf. zum Unterbrechen bzw. Fortsetzen von Tasks führt.

2 Fiber Pool verwenden

In diesem Kapitel wird die Verwendung von Fiber Pool und der dazugehörigen Klassenbibliothek beschrieben.

2.1 Fiber Pool für ein Projekt einrichten

Für die Verwendung von Fiber Pool in einem Projekt sind die folgenden beiden Schritte erforderlich:

1. Erstellen der FiberPoolAPI-Bibliotheken
2. Konfiguration der Projekteigenschaften

Der erste Schritt entfällt, falls das Projekt die Klassenbibliothek nicht benötigt.

Die folgenden Abschnitte beschreiben die einzelnen Schritte unter Verwendung von Microsoft® Visual Studio® 2008.

2.1.1 Die FiberPoolAPI-Bibliotheken erstellen

Um die Fiber Pool-Klassenbibliothek verwenden zu können, müssen Sie sie zunächst erstellen:

1. Öffnen Sie die Projektmappen-Datei „FiberPoolAPI.sln“, die im Entwicklungspaket enthalten ist.
2. Passen Sie die Projekteinstellungen der Bibliothek mit den Einstellungen Ihres Projekts an. Dies sind u.a. die Auswahl der Laufzeitbibliothek und die C++-Sprachfeatures.
3. Erstellen Sie die Bibliothek für die gewünschte Konfiguration.

Die Projektdatei enthält Konfigurationen für 32- und 64-Bit, jeweils als Debug- und als Release-Version.

Die erstellten statischen Bibliotheken werden im Verzeichnis „lib“ erstellt, mit den folgenden Bezeichnungen:

- FiberPoolAPI32.lib für die 32-Bit-Release-Konfiguration
- FiberPoolAPI32d.lib für die 32-Bit-Debug-Konfiguration
- FiberPoolAPI64.lib für die 64-Bit-Release-Konfiguration
- FiberPoolAPI64d.lib für die 64-Bit-Debug-Konfiguration

2.1.2 Die Projekteigenschaften konfigurieren

Folgende Projekteigenschaften müssen Sie für die Verwendung von Fiber Pool anpassen:

1. **Include-Pfad:** Fügen Sie das Verzeichnis „include“ als zusätzliches Include-Verzeichnis über die Compiler-Einstellungen in Ihr Projekt hinzu. Alternativ kann das Verzeichnis auf für alle Projekte in den IDE-Einstellungen bereitgestellt werden.
Zur Verwendung der Headerdateien genügt es, die Datei „FiberPoolAPI.H“ zu laden, die mit sämtlichen anderen Headerdateien verknüpft ist. Wird die Klassenbibliothek nicht benötigt, so kann das Laden der zugehörigen Headerdateien durch Setzen der Präprozessorrichtlinie `FIBERPOOL_NOAPI` verhindert werden.
2. **Bibliotheken:** Fügen Sie dem Linker die Datei „lib\FiberPool32.lib“ für 32-Bit-Projekte bzw. „lib\FiberPool64.lib“ als Eingabedatei hinzu.
Für die Verwendung der Klassenbibliothek fügen Sie zusätzlich für die jeweilige Konfiguration die entsprechende statische Bibliothek hinzu.

Die Fiber Pool-DLLs liegen im Verzeichnis „bin“. Erweitern Sie dazu die PATH-Umgebungsvariable und fügen dieses Verzeichnis hinzu.

2.2 Konzeptionelle Einschränkungen

Der von Fiber Pool taskbasierte Ansatz kapselt die Thread- und Fiber-Verwaltung und erleichtert somit die Programmierung komplexer, skalierbarer Algorithmen.

Durch diese Kapselung wird es dem Scheduler ermöglicht, Tasks lastbalanciert auf gerade verfügbare Threads auszuführen. D.h., dass Programmteile einer Task während ihrer Laufzeit auf verschiedene Threads ausgeführt werden können.

Aus diesem Grund muss bei der Programmierung von Tasks auf die Verwendung von **thread-affinen oder thread-bezogenen** Funktionen oder Objekten verzichtet werden. Dies schließt die Verwendung von **COM** mit ein.

Eine weitere Einschränkung existiert bei blockierenden Systemaufrufen: Die Verwendung ist grundsätzlich möglich, bei einem Blockieren des Aufrufs führt dies jedoch zu einer Reduzierung der Performance, weil der Thread, der die Task bearbeitet, ebenfalls blockiert und keine weitere Tätigkeit durchführen kann.

Wenn möglich, sollten blockierende Systemaufrufe durch nicht-blockierende Aufrufe ersetzt werden.

2.3 Erste Schritte

Dieser Abschnitt beschreibt die grundlegende Verwendung von Fiber Pool.

2.3.1 Einen Fiber Pool initialisieren

Ein Fiber Pool-Objekt wird mit der **InitializeFiberPool**-Funktion erzeugt. Als Argument erwartet sie einen Zeiger auf eine **FiberPoolInfo**-Struktur, die die Eigenschaften des Fiber Pools beschreibt, z.B. die Anzahl der Threads oder die Affinitätsmaske der zu verwendenden Prozessoren.

Bei erfolgreicher Erzeugung gibt die Funktion einen Zeiger auf die **IFiberPool**-Schnittstelle zurück, über diese die weitere Kommunikation erfolgt. Eine Speicherung des Zeigers für Tasks ist nicht notwendig, da Tasks den Zeiger jederzeit über die Funktion **GetFiberPool** erhalten können.

Zum Beenden und Freigeben des Fiber Pools muss die Methode **IFiberPool::Release** aufgerufen werden.

Im folgenden Beispiel wird ein Fiber Pool mit Default-Werten initialisiert:

```
// Initialisieren der FiberPoolInfo-Struktur
FiberPoolInfo fiberPoolInfo = { 0 };
fiberPoolInfo.version = FIBERPOOLVERSION; // Version der Fiber Pool-
                                           // Schnittstelle

// Initialisieren des Fiber Pools
IFiberPool* fiberPool = InitializeFiberPool(&fiberPoolInfo);
```

2.3.2 Den Scheduler starten und beenden

Nach erfolgreicher Initialisierung des Fiber Pools kann sein Scheduler mit der Methode **IFiberPool::Run** gestartet werden. Erst durch diesen Aufruf werden die Threads erzeugt und es wird mit der Verarbeitung von Tasks begonnen.

Die Verarbeitung wird mit der Methode **IFiberPool::Release** beendet: Existierende Tasks werden noch verarbeitet, untätige Threads werden jedoch nach und nach beendet.

Beispiel:

```
// Initialisieren der FiberPoolInfo-Struktur
FiberPoolInfo fiberPoolInfo = { 0 };
fiberPoolInfo.version = FIBERPOOLVERSION; // Version der Fiber Pool-
                                           // Schnittstelle

// Initialisieren des Fiber Pools
IFiberPool* fiberPool = InitializeFiberPool(&fiberPoolInfo);

// Fiber Pool starten
fiberPool->Run();

// Fiber Pool verwenden...

// Fiber Pool beenden
fiberPool->Release();
```

2.3.3 Einen Thread für die Kommunikation mit Tasks einrichten

Grundsätzlich ist es nicht möglich, Methoden einer Task von einem Thread außerhalb des Fiber Pools aus aufzurufen, weil Tasks einen Fiber-Kontext benötigen, um über die Schnittstellen **IFiberPool** und **IFiberControl** mit dem Fiber Pool kommunizieren zu können.

Mit der Methode **IFiberPool::ActivateOnThread** erzeugt der Fiber Pool ein Hilfsobjekt für den aktuellen Thread, über dieses Task-Methoden, die von einem Thread aufgerufen werden, eingeschränkten Zugriff auf die Fiber Pool-Schnittstellen erhalten.

Die Funktionen **GetFiberPool**, **GetFiberControl** und **GetFiberTask** verweisen dann auf dieses Hilfsobjekt, wenn sie von einem Thread aus aufgerufen werden. Eine spezielle Implementierung der **IFiberControl**-Schnittstelle ermöglicht dabei auch die Synchronisation zwischen Threads und Tasks.

Das Hilfsobjekt wird über die Methode **IFiberPool::DeactivateOnThread** gelöscht.

2.3.4 Eine Task implementieren

Tasks müssen die **IFiberTask**-Schnittstelle implementieren, um vom Fiber Pool Scheduler ausgeführt werden zu können. Die Klassenbibliothek stellt dazu die Basisklasse **FiberTaskBase** zur Verfügung, die bereits ein Standardverhalten für die meisten Methoden implementiert, so dass lediglich die **IFiberTask::Run**-Methode bereitgestellt werden muss:

Beispiel:

```
// Die Hello World-Task
class HelloWorldTask : public FiberTaskBase
{
public:
    void FIBERPOOLCALLTYPE Run()
    {
        cout << „Hello World!“ << endl;
    }
};
```

Nach Beendigung einer Task kann sich das Task-Objekt selbst löschen, wenn man die **ITaskFinishNotify::OnTaskFinished**-Methode wie folgt überlädt:

```
void FIBERPOOLCALLTYPE OnTaskFinished()
{
    __super::OnTaskFinished();
    delete this;
}
```

Die Basisimplementierung dieser Methode benachrichtigt das Objekt, das über **IFiberTask::SetNotify** gesetzt wurde, über die Beendigung der Task.

Zur effizienten Speicherung unterbrochener Tasks ermöglicht die **IFiberTask::GetCookieListEntry**-Methode die Bereitstellung einer im Task-Objekt instanziierten Datenstruktur, die Container (z.B. Listen) zur Verwaltung ihrer Elemente nutzen können, ohne dynamisch dafür Speicher allokiert zu müssen.

Die Struktur **CookieListEntry** ist in der Klassenbibliothek definiert, kann aber für eigene Zwecke angepasst werden. Ihre maximale Größe kann **18 * sizeof(void*)** sein.

Die Fortsetzung unterbrochener Tasks wird üblicherweise vom Fiber Pool Task Scheduler gesteuert. Die Steuerung kann durch Überladen der Methode **IFiberTask::GetFiberControl** auf eigene Bedürfnisse angepasst werden. Zum Beispiel verhindert die **VolumeIOManager**-Klasse der Klassenbibliothek durch eine eigene Steuerung den parallelen Zugriff auf Dateien einer Festplatte durch mehrere Tasks.

2.3.5 Eine Task ausführen

Eine Task wird über **IFiberPool::AddTask** beim Scheduler zur Ausführung eingereicht. Den Zeiger auf die **IFiberPool**-Schnittstelle erhalten Tasks durch Aufrufen der **GetFiberPool**-Funktion. Threads, die für die Kommunikation mit Tasks eingerichtet wurden, erhalten den Zeiger ebenfalls über die **GetFiberPool**-Funktion. Allen anderen Threads muss der Zeiger, der von **InitializeFiberPool** zurückgegeben wurde, als Argument übergeben werden.

Die Ausführung der Task erfolgt asynchron, so dass ihr Ergebnis nicht sofort vorliegt. Ist das Vorliegen des Ergebnisses an einer Programmstelle für die weitere Fortführung notwendig, so muss auf die Beendigung der Task gewartet werden.

Die Klassenbibliothek bietet dafür eine Vielzahl an Möglichkeiten, die im nächsten Abschnitt beschrieben werden.

2.4 Synchronisationsobjekte

Die Klassenbibliothek stellt mehrere Klassen bereit, die Tasks für verschiedene Synchronisationsszenarien einsetzen können.

2.4.1 Programmbereiche für exklusiven Zugriff sperren

Der exklusive Zugriff auf eine Ressource oder einen Programmbereich kann mit **CriticalSection** gesteuert werden.

Mit **CriticalSection::Enter** erhält eine Task exklusiven Zugriff auf den Bereich. Der Aufruf blockiert, wenn eine andere Task den exklusiven Zugriff bereits erhalten hat.

Der exklusive Zugriff wird mit **CriticalSection::Leave** freigegeben. Beide Methoden **CriticalSection::Enter** und **CriticalSection::Leave** können von einer Task rekursiv aufgerufen werden.

Bei sehr kurzen Zugriffen kann statt **CriticalSection::Enter** die Methode **CriticalSection::BusyEnter** aufgerufen werden. Dies kann zu einer Leistungssteigerung führen, wenn eine baldige Erteilung des Zugriffs erwartet werden kann.

Eine Task kann mit **CriticalSection::TryEnter** versuchen, Zugriff zu erhalten, ohne zu blockieren.

Zum automatischen Aufrufen von **CriticalSection::Enter** und **CriticalSection::Leave** kann für einen Sichtbarkeitsbereich die Klasse **CriticalSectionLock** verwendet werden.

Beispiel:

```
// Implementierung einer List-Klasse für parallelen Zugriff
template <typename TYPE> class SharedList
{
public:
    void push_back(const TYPE& item)
    {
        CriticalSectionLock lock(m_cs);
        m_list.push_back(item);
    }

    void pop_back()
    {
        CriticalSectionLock lock(m_cs);
        m_list.pop_back();
    }

    ...

private:
    CriticalSection m_cs;
    std::list<TYPE> m_list;
};
```

Eine für Lese- und Schreibzugriffe flexiblere Synchronisationsklasse als **CriticalSection** ist **RWLock**.

RWLock stellt sicher, dass sich in den Programmbereichen entweder nur genau eine schreibende Task oder eine beliebige Anzahl an lesenden Tasks befinden.

Lesezugriffe werden über die Methoden **RWLock::AcquireReadLock** und **RWLock::ReleaseReadLock** gesteuert, Schreibzugriffe über **RWLock::AcquireWriteLock** und **RWLock::ReleaseWriteLock**.

Zum automatischen Aufrufen dieser Methoden kann für einen Sichtbarkeitsbereich die Klasse **ReadLock** bzw. **WriteLock** verwendet werden.

Beispiel:

```
// Implementierung einer Value-Klasse für parallelen Zugriff
template <typename TYPE> class SharedValue
{
public:
    void operator=(const TYPE& value)
    {
        WriteLock lock(m_lock);
        m_value = value;
    }

    operator TYPE() const
    {
        ReadLock lock(m_lock);
        return m_value;
    }

    ...

private:
    mutable RWLock m_lock;
    TYPE          m_value;
};
```

2.4.2 Ereignisbehandlung

Tasks können mit der **Event**- bzw. **LocalEvent**-Klasse auf Ereignisse warten, die in anderen Tasks ausgelöst werden.

Auf **Event**-Objekte können mehrere Tasks auf das Auslösen eines Ereignisses warten, während bei **LocalEvent** nur eine Task darauf warten kann.

Ein Ereignis wird mit **Event::Set** ausgelöst. Tasks können mit **Event::Wait**, **Event::TryWait** oder **Event::BusyWait** auf das Ereignis warten.

LocalEvent kann im Gegensatz zu **Event** innerhalb eines Sichtbarkeitsbereichs eingesetzt werden, z.B.:

```
// Eine Funktion, die eine Task ausführen lässt und auf sie wartet
void MyFunction()
{
    LocalEvent event;
    // Eine Task, die beim Beenden das Ereignis auslöst
    IFiberTask* task = new MyTask(event);
    ...
    event.Wait()
}
```

Durch die C++-Implementierung ist die Verwendung von **Event** hier nicht möglich, da ihre Instanz beim Verlassen des Sichtbarkeitsbereichs gelöscht wird, obwohl sie in einer anderen Task evtl. noch mit der Freigabe wartender Tasks beschäftigt ist.

2.4.3 Zugriff auf Ressourcen begrenzen

Mit der **Semaphore**-Klasse ist es möglich, den Zugriff auf eine Ressource für eine bestimmte Anzahl an Nutzern zu begrenzen.

Die maximale Anzahl an Nutzern wird im Konstruktor übergeben. Mit **Semaphore::Wait**, **Semaphore::TryWait** oder **Semaphore::BusyWait** erhält man Zugriff auf die Ressource. Der Zugriff wird mit **Semaphore::Release** wieder freigegeben.

Zum automatischen Aufrufen von **Semaphore::Wait** und **Semaphore::Release** kann für einen Sichtbarkeitsbereich die Klasse **SemaphoreLock** verwendet werden.

2.4.4 Gruppen

Mehrere Tasks können zu Gruppen zusammengefasst werden, sodass auf ihre gemeinsame Beendigung gewartet werden kann. Hierfür werden die Klassen **TaskGroup** und **RestrictedTaskGroup** bereitgestellt. Beide sind von der Klasse **TaskGroupBase** abgeleitet.

Tasks werden über **TaskGroupBase::AddTask** hinzugefügt. Auf deren Beendigung wird mit **TaskGroupBase::Wait**, **TaskGroupBase::TryWait** oder **TaskGroupBase::BusyWait** gewartet.

Grundsätzlich werden hinzugefügte Tasks direkt an den Fiber Pool Scheduler weitergeleitet. Um die Anzahl der asynchron auszuführenden Tasks zu begrenzen, kann dafür in **RestrictedTaskGroup** ein maximaler Wert festgelegt werden. Beim Erreichen dieses Werts werden neu hinzugefügte Tasks dann nicht mehr an den Scheduler weitergeleitet, sondern synchron ausgeführt.

2.4.5 Transitionen

Zum Ausführen bzw. Fortsetzen von Tasks in Abhängigkeit von einem oder mehreren Ereignissen kann die **Transition**-Klasse verwendet werden.

Die Anzahl der Ereignisse wird im Konstruktor gesetzt. Mit **Transition::IncrementActivations**, **Transition::operator ++**, **Transition::AddActivations** oder **Transition::operator +=** kann die Zahl angepasst werden.

Ein Ereignis wird mit **Transition::Set** ausgelöst. Dabei wird der Ereigniszähler um 1 reduziert. Erreicht dieser den Wert 0, so werden alle mit dem Transition-Objekt gebundenen Tasks freigegeben.

Neue Tasks werden über **Transition::AddTask** hinzugefügt. Sie werden erst dann an den Scheduler übergeben, wenn alle Ereignisse ausgelöst wurden.

Bestehende Tasks können auf die Transition mit **Transition::Wait**, **Transition::TryWait** oder **Transition::BusyWait** warten.

2.4.6 Auf die Berechnung einer Variablen warten

Tasks können auf die Berechnung von Variablen aus anderen Tasks durch Verwenden der **Unassigned<T>**-Klasse warten. Dies geschieht automatisch beim Zugriff durch **Unassigned<T>::operator T**.

Sobald der Wert durch **Unassigned<T>::operator =** gesetzt wird, werden alle wartenden Tasks freigegeben.

2.4.7 Eigene Synchronisationsklassen entwickeln

Sind spezielle Anforderungen für die Synchronisation zwischen Tasks gegeben, die nicht durch die bereitgestellten Klassen abgedeckt werden, so können eigene Klassen unter Verwendung des Fiber Pool API oder der Fiber Pool Synchronization Library entwickelt werden.

Aufgaben einer Synchronisationsklasse sind:

- Speichern eines Anwendungszustands
- Prüfen des Anwendungszustands, um
- ggf. eine Task zu blockieren, oder
- ggf. eine Task fortzusetzen.
- Speichern von blockierten Tasks

Synchronisationsklassen mit dem Fiber Pool API

Das API bietet für die Entwicklung eigener Synchronisationsklassen lediglich Unterstützung zum Blockieren und Fortsetzen von Tasks.

Anhand eines Beispiels werden die notwendigen Schritte zur Implementierung einer Synchronisationsklasse beschrieben.

Das folgende Beispiel ist ein Zähler, der alle zugreifenden Tasks solange blockiert, bis ein bestimmter Wert überschritten wird.

Die Grundstruktur ist:

```
class Counter : public IFiberWaitCallback
{
public:
    Counter(long value) : m_value(value), m_currentValue(0) {}

private:
    const long m_value;
    volatile long m_currentValue;
};
```

Die Vererbung von **IFiberWaitCallback** wird weiter unten beschrieben.

Das Speichern des Anwendungszustands erfolgt in der Variablen ‚m_currentValue‘. Da Tasks den Zustand parallel aus verschiedenen Threads ändern können, muss sie mit dem Schlüsselwort ‚volatile‘ deklariert werden.

Als Nächstes muss die Klasse eine Methode bereitstellen, die Tasks blockiert, wenn die Zustandsbedingung nicht erfüllt ist:

```
void Counter::Wait()
{
    if (m_currentValue <= m_value)
        GetFiberControl()->Wait(this, 0);
}
```

Die Methode **IFiberControl::Wait** erwartet als erstes Argument einen Zeiger auf eine **IFiberWaitCallback**-Schnittstelle. Über sie benachrichtigt der Scheduler, dass das Blockieren der Task abgeschlossen ist. Erst nach dieser Benachrichtigung kann die blockierte Task im Synchronisationsobjekt gespeichert werden.

Folgende Aufgaben werden in **IFiberWaitCallback::WaitCallback** durchgeführt:

- Speichern der blockierten Task
- Erneute Prüfung des Anwendungszustands, um ggf. blockierte Tasks fortzusetzen

Wegen der notwendigen Prüfung des Anwendungszustands ist es sinnvoll, dass die Implementierung von **IFiberWaitCallback** in der Synchronisationsklasse enthalten ist.

Zur Speicherung blockierter Tasks sollte eine nicht-blockierende (sog. „lock-free“) Implementierung eines Containers verwendet werden. Fiber Pool stellt hierfür die Klasse **LIFOQueue** bereit.

Für die Weiterentwicklung des Beispiels wird nun angenommen, dass die Klasse eine Variable ‚m_taskQueue‘ vom Type **LIFOQueue** enthält:

```
void Counter::WaitCallback(void* cookie, void* args)
{
    CookieListEntry* cle = GetFiberTask(cookie)->GetCookieListEntry();
    cle->cookie = cookie;
    m_taskQueue.push((LIFOQueueItem*)cle);
    if (m_currentValue > m_value)
        ResumeTasks();
}
```

In der ersten Zeile wird aus der blockierten Task die Datenstruktur geholt, die für die Speicherung der Task in einem Container vorgesehen ist. Durch die Bereitstellung dieser Struktur durch Tasks entfällt der Erstellungsaufwand auf Seiten des Containers.

In Fiber Pool handelt es sich um die Struktur **CookieListEntry**, die für die Verwendung mit **LIFOQueue** abgestimmt ist.

In der zweiten Zeile wird die blockierte Task im Container gespeichert.

Zwischen dem Aufruf von **IFiberControl::Wait** und dem Speichern der Task kann sich der Anwendungszustand geändert haben. Daher ist es notwendig, diesen Zustand zu überprüfen und ggf. die blockierten Tasks freizugeben. Dies geschieht in den letzten beiden Zeilen.

Die Methode ‚ResumeTasks‘ kann wie folgt implementiert werden:

```
void Counter::ResumeTasks()
{
    CookieListEntry* cle;
    while (cle = (CookieListEntry*)m_taskQueue.pop())
        GetFiberTask(cookie) -> GetFiberControl() -> Resume(cle->cookie);
}
```

Die blockierten Tasks werden nach und nach aus dem Container geholt und fortgesetzt.

Zum Abschluss ist noch eine Methode zu implementieren, mit der der Anwendungszustand geändert werden kann:

```
void Counter::Increment()
{
    if (_InterlockedIncrement(&m_currentValue) > m_value)
        ResumeTasks();
}
```

Der geänderte Zustand wird sofort überprüft und blockierte Tasks werden ggf. fortgesetzt.

Synchronisationsklassen mit der Fiber Pool Synchronization Library

Wesentlich einfacher als mit dem Fiber Pool API ist die Entwicklung eigener Synchronisationsklassen unter Verwendung der Klassenbibliothek. Die dafür bereitgestellten Klassen übernehmen die gesamte Taskverwaltung, so dass nur noch ein kleiner Teil selbst implementiert werden muss.

Folgende Basisklassen werden bereitgestellt:

- **SingleBlockObject<>**, zum Erstellen von Synchronisationsklassen, die nur eine Task blockieren können.
- **MultiBlockObject**, zum Erstellen von Synchronisationsklassen, die mehrere Tasks blockieren können.
- **WaitObject** und davon abgeleitet **SingleWaitObject<>** und **MultiWaitObject**, zum Erstellen von Synchronisationsobjekten mit Wartesemantik.

Das vorherige Beispiel ‚Counter‘ kann durch Verwenden der Basisklassen wesentlich vereinfacht werden:

```

class Counter : public MultiWaitObject
{
public:
    Counter(long value) : m_value(value), m_currentValue(0) {}

    void Increment()
    {
        if (_InterlockedIncrement(&m_currentValue) > m_value)
            Unblock();
    }

    // Aus der Basisklasse WaitObject
    bool TryWait()
    {
        return m_currentValue > m_value;
    }

private:
    const    long m_value;
    volatile long m_currentValue;
};

```

Eigene Synchronisationsklassen aus bestehenden Klassen entwickeln

Noch einfacher ist die Entwicklung eigener Synchronisationsklassen, wenn als Basis bestehende Klassen verwendet werden können.

Für das Beispiel ‚Counter‘ könnte man als Basis die Klasse **Event** einsetzen:

```

class Counter : Event
{
public:
    Counter(long value) : m_value(value), m_currentValue(0) {}

    void Increment()
    {
        if (_InterlockedIncrement(&m_currentValue) > m_value)
            Set();
    }

    Event::Wait;
    Event::TryWait;
    Event::BusyWait;

private:
    const    long m_value;
    volatile long m_currentValue;
};

```

2.5 Virtueller Speicher

Ein häufiger Anwendungsfall ist das sequentielle Erzeugen und Verarbeiten von Daten, z.B. das Lesen einer Datei in den Speicher und die Berechnung ihrer Prüfsumme.

Die Klassenbibliothek stellt dafür folgende Synchronisationsklassen bereit, die die Steuerung der Tasks und die Verwaltung des Speichers übernehmen:

- **SequentialVirtualMemoryPipe** für eine schreibende und eine lesende Task
- **SequentialVirtualMemoryTee** für eine schreibende und mehreren lesenden Tasks

Um den Speicherverbrauch zu begrenzen, kann dem Synchronisationsobjekt für den schreibenden Zugriff eine Speicherverwaltungsstrategie festgelegt werden:

- **MM_Ignore:** Mit dieser Strategie wird keine Überprüfung des Speicherverbrauchs durchgeführt. Die schreibende Task erhält immer den angeforderten Speicher und wird nicht blockiert.
- **MM_PageLimit:** Die schreibende Task kann bis zu einer bestimmten Größe Speicher anfordern. Ist dieser aufgebraucht, so wird die Task solange blockiert, bis lesende Tasks genügend Speicher freigegeben haben.
- **MM_System:** Die schreibende Task kann solange Speicher anfordern, bis der freie physische Speicher unter eine bestimmte Grenze fällt. Danach wird die Task blockiert und muss warten, bis genügend Speicher von lesenden Tasks freigegeben wird. Um einen Fortschritt auch bei nicht verfügbarem Speicher zu ermöglichen, kann eine Minimalgröße an Speicher festgelegt werden, die der schreibenden Task garantiert wird.

Speicher wird nach und nach freigegeben, wenn sämtliche lesenden Tasks einen Bereich verlassen haben.

Mit **ISequentialVirtualMemory::Reserve** wird ein Speicherbereich reserviert. Lesende Tasks erhalten die Größe des Bereichs über **ISequentialVirtualMemory::ReservedSize**. Dieser Aufruf kann die aufrufende Task blockieren, wenn noch kein Speicherbereich reserviert wurde.

Lesende Tasks, die schon von Beginn an einen reservierten Speicherbereich benötigen, sollten nicht direkt an den Fiber Pool Task Scheduler, sondern an das **Transition**-Objekt des Speicherobjekts übergeben werden. Dadurch wird deren Ausführung bis zur Reservierung des Speichers hinausgezögert. Das **Transition**-Objekt erhält man über die Methode **GetMemoryReservedTransition**.

Über die **ISequentialVirtualMemoryWriter**-Schnittstelle, deren Zeiger man über **ISequentialVirtualMemory::GetWriter** erhält, kann die schreibende Task auf den Speicher zugreifen. Über **ISequentialVirtualMemoryWriter::GetWriteMemory** wird Speicher zum Schreiben angefordert. Nach dem Schreiben kann der Speicher über **ISequentialVirtualMemoryWriter::MakeWriteMemoryValid** zum Lesen freigegeben

werden. Alternativ können beide Schritte beim Kopieren aus einem anderen Speicherbereich über **ISequentialVirtualMemoryWriter::Write** ausgeführt werden.

Gemäß der festgelegten Speicherverwaltungsstrategie wird bei blockierendem Aufrufen dieser Methoden die schreibende Task blockiert, wenn der angeforderte Speicher nicht verfügbar ist.

Den Zeiger auf die **ISequentialVirtualMemoryReader**-Schnittstelle erhalten lesende Tasks über **ISequentialVirtualMemory::GetReader**. Speicher zum Lesen wird über **ISequentialVirtualMemoryReader::GetReadMemory** angefordert und nach seiner Bearbeitung über **ISequentialVirtualMemoryReader::MarkRead** freigegeben. Alternativ können beide Schritte über **ISequentialVirtualMemoryReader::Read** beim Kopieren ausgeführt werden.

Mit **ISequentialVirtualMemoryReader::Touch** kann ein Speicherbereich zum Lesen angefordert werden, ohne die Leseposition zu verändern. Speicherbereiche können über **ISequentialVirtualMemoryReader::Seek** übersprungen werden.

Blockierende Aufrufe dieser Methoden blockieren die lesenden Tasks, wenn der angeforderte Speicher nicht verfügbar ist.

Bei Verwendung von **SequentialVirtualMemoryTee** muss nach Verwendung eines Leseobjekts die Methode **ISequentialVirtualMemory::NotifyReadingCompleted** aufgerufen werden, um die Leseposition an das Ende des Speichers zu setzen.

2.6 Dateien

Für den Zugriff auf Dateien stellt die Klassenbibliothek die Klasse **File** zur Verfügung. Sie bietet Methoden sowohl für synchrones als auch asynchrones Lesen und Schreiben. Ihre Implementierung verwendet dazu ausschließlich nicht-blockierende Systemfunktionen, um die Threads des Fiber Pool Task Schedulers nicht zu blockieren.

Dateien werden mit **File::Create** bzw. **File::CreateNew** geöffnet. Die zweite Methode öffnet die Datei mit erweiterten Privilegien, um eine bessere Schreibperformance zu erzielen.

Nach dem Öffnen können die synchronen bzw. asynchronen Versionen von **File::Read** oder **File::Write** aufgerufen werden. Das Schließen einer Datei erfolgt über **File::Close**.

Zum parallelen Arbeiten mit mehreren Dateien kann der File I/O Scheduler eingesetzt werden. Er sorgt dafür, dass Dateioperationen auf einer Festplatte nur von einer Task zur gleichen Zeit ausgeführt werden, um ein ständiges Verschieben des Lese-/Schreibkopfs zu vermeiden.

Aktuell bietet der File I/O Scheduler folgende Funktionen an:

- Lesen einer Verzeichnisstruktur
- Einlesen einer Datei in den Speicher

- Schreiben einer Datei aus dem Speicher
- Ausführen einer benutzerdefinierten I/O-Task