Praktikum IPCPool

Task Queues und Thread Pools

Frühlingssemester 2016

M. Thaler

Überblick

In diesem Praktikum lernen sie mit POSIX Message Queues umzugehen und für verschiedene Anwendungsfälle einzusetzen. Gleichzeitig lernen sie kennen, wie Tasks Queues und Thread Pools arbeiten und implementiert werden können und erhalten ein vertieftes Verständnis für Threading Konzepte.

Im ersten Teil des Praktikums setzen sie Messages Queues für den gegenseitigen Austausch von Information zwischen zwei Prozessen ein.

Im zweiten Teil implementieren sie als Anwendung einen einfachen Thread Pool mit Task Queue, der mit Hilfe einer bzw. mehreren Message Queues implementiert wird. Dabei beschäftigen sie auch auch mit Funktionen zur Steuerung von Threads (Thread Cancellation).

Inhaltsverzeichnis

1 Einführung

Der Datenaustausch zwischen kooperierenden Tasks mit Messages hat den Vorteil, dass implizit eine Message Queue zur Verfügung gestellt wird und Meldungen vom Empfänger asynchron abgeholt werden können. Zudem blockieren Empfänger solange die Message Queue leer ist. Diese Eigenschaften werden wir hier nutzen um einen einfachen Thread Pool mit Task Queue zu implementieren.

Wir werden POSIX Message Queues verwenden und stellen ihnen Programmrahmen für die einzelnen Aufgabenstellungen zur Verfügung. Im Text geben wir jeweils an, welche POSIX Funktionen verwendet werden sollen, die Details zu den Funktionen finden sie in den jeweiligen Manuals mit "`man Funktionsname`". Eine Übersicht zu den POSIX Message Queues finden sie mit "`man mq_overview`".

2 POSIX Message Queues

In dieser Aufgabe implementieren sie den Austausch von Messages zwischen zwei Prozessen. Ein Prozess sendet dabei den String "Ping" an den Empfänger Prozess, der ihn ausgibt und anschliessend den String "Pong" an den Sender zurückschickt, der ihn ausgibt ... etc.

Im Verzeichnis "`a1`" finden sie Programmrahmen für die zwei Prozesse `main.c` und `child.c` implementiert. In `msgDefs.h` sind die Strings für die Queue Namen definiert sowie die maximale Anzahl Messages in der Queue: `MAX_MSGS`.

2.1 Aufgabe 1

Implementieren sie das Hauptprogramm in `main.c` und den Kind Prozess in `child.c`. Gehen sie dabei nach folgender Anleitung vor:

1. Löschen Sie im Hauptprogramm zuerst die Message Queues mit `mq_unlink()` Damit stellen sie sicher, dass keine "Überbleibsel" der Queues vorhanden sind, was manchmal bei einem Programmabbruch vorkommen kann.

2. Setzen sie nun die Attribute in der Struktur `struct mq_attr attr` auf die in `msgDefs.h` vordefinierten Wert. Die Struktur `mq_attr` ist im Manual zu `mq_getattr` beschrieben. Die vom Betriebssystem definierten Werte der Attribute sind im "/proc" File System abgelegt, die entsprechenden Pfade finden sie in den Kommentaren in `msgDefs.h`.

3. Öffnen sie die beiden Queues mit `mq_open()`, verwenden sie die in `msgDefs.h` definierten Namen. Als Flags benötigen Sie `O_CREAT` und `O_RDWR` und für die Berechtigung `0700` (führende `0` bedeutet Oktalzahl). Verwenden sie zudem die oben beschriebenen Attribute. Sie können für beide Queues die gleiche Attribut Struktur verwenden (Werte werden kopiert).

4. Für das Lesen aus einer Queue muss ein Buffer zur Verfügung gestellt werden: die Grösse dieses Buffers muss gleich dem Attribut `mq_msgsize` gewählt werden. Abfragen können sie das Attribut mit `mq_getattr()`. Der Buffer kann anschliessend z.B. mit `char buf[size]` deklariert werden: aktuelle C Implementation erlauben Variablen als Arraygrössen zu verwenden.

5. Implementieren sie nun das Senden und Empfangen der Meldungen innerhalb der for-Schleife mit `mq_send()` und `mq_receice()`.

6. Nach Beendigung der for-Schleife in `main()` muss eine Meldung der Länge 1 mit dem Zeichen `'\0'` (dezimal `0`) gesendet werden: damit wird dem Kind Prozess signalisiert, dass er terminieren soll (siehe dazu auch Abschnitt 2.2).

7. Zuletzt müssen die Queues mit `mq_close()` und geschlossen und mit `mq_unlink()` gelöscht werden.

Gehen sie bei der Implementation des Kind-Prozesses (`child.c` genau gleich vor, allerdings dürfen sie hier `mq_unlink()` nicht verwenden und die Attribute müssen nicht übergeben werden.

2.2 Terminierung Child

Im Kind-Prozess werden die Messages innerhalb einer `while(1)`-Schleife empfangen und gesendet. Damit das Kind diese Schleife verlässt, erhält es vom Hauptprogramm eine Meldung , die als erstes Zeichen ein `'\0'`-Character (dezimal `0`) enthält, die Länge der Meldung spielt dabei keine wesentliche Rolle, wird aber sinnvollerweise als `1` gewählt.

2.3 Testing

Das Programm läuft korrekt, wenn das Programm abwechselnd Ping und Ping ausgibt.

3 Task Queue und Thread Pool

In folgenden werden wir einen einfachen Thread Pool mit einer Task Queue implementieren, wobei die Task Queue mit Hilfe einer POSIX Message Queue implementiert wird. Dabei holen sich die Threads Task um Task aus der Queue und arbeiten sie ab. Die Threads müssen nur einmal beim Programmstart erzeugt werden und stehen dann während der gesamten Laufzeit zur Verfügung. Dies ist einerseits schneller als wenn für jeden Task ein Thread gestartet werden muss, andererseits werden die Tasks automatisch parallel ausgeführt, ohne dass sich der Anwender darum kümmern muss. Zudem kann die Anzahl Threads auf die Anzahl Cores des Prozessor abgestimmt werden.

Die meisten modernen Programmiersprachen Unterstützen heute Thread Pools mit entsprechenden Bibliotheken, in Java z.B. das Executor Framework.

3.1 Tasks

In diesem Praktikum werden wir nur Tasks verwenden, die einmal gestartet zu Ende laufen und nicht unterbrochen werden können. Blockiert zudem ein Task, blockiert auch der Thread der ihn ausführt. Task werden ähnlich wie Threads mit einer Funktion definiert.

3.1.1 Das Task Objekt

Um die Handhabung von Tasks zu erleichtern, arbeiten wir mit Task Objekten. Ein Task Objekt ist eine Struktur, die mindestens zwei Einträge hat und am besten mit Hilfe einer Typendefinition `typedef` beschrieben werden:

```
typdef struct taskObjectFoo {
    void *this;
    void (*fun)(void *);
    // ....
} tObjFoo_t;
```

Der erste Eintrag ist ein Pointer auf die Struktur selbst, der zweite Eintrag die Task Funktion mit Signatur "`void fun(void *)`". Zum Starten eines Tasks ruft ein Thread des Pools die Funktion `fun` auf übergibt `this` als Parameter.

Die Struktur kann beliebig erweitert werden, z.B. mit einem Feld für Rückgabewerte, etc., einzige Bedingung: die beiden ersten Einträge müssen obiger Definition entsprechen. Mit Hilfe dieser Konvention kann ein Task sehr einfach über eine Meldung in die Task Queue eingereiht werden (Details siehe unten).

Hinweis: für verschiedene Tasks müssen auch verschiedene Task Objekte definiert werden.

3.2 Die Thread Pool Funktionen

Im Verzeichnis "a2" finden sie das Hauptprogramm (`main.c` mit einer einfachen Testanwendung sowie den Programmrahmen für die Implementierung des Thread Pools (`threadpool.c`, `threadpool.h`).

Für die Implementierung des Thread Pools werden drei Funktionen benötigt, die in `threadpool.h` definiert sind:

- `void initThreadPool(int nThreads)`
- `void stopThreadPool(void)`
- `void startTask(void *taskObject)`

In `threadpool.h` ist ebenfalls ein Typ für die Taskfunktion definiert: **`taskfun_t`**.

Hinweis: implementieren sie sämtliche weiteren Definitionen, wie Queue Namen, etc. in `threadpool.c`. Damit bleibt die Schnittstelle des Thread Pools minimalistisch und gewährt optimales "Hiding". Verwenden sie aus diesen Gründen auch das Attribut `static` für globale Variablen.

3.2.1 Funktion: `void initThreadPool(int nThreads)`

Die Funktion `initThreadPool()` setzt zuerst eine Message Queue für die Realisierung der Task Queue auf (siehe Abschnitt 2) und startet anschliessend die entsprechende Anzahl Threads mit `pthread_create()` und der Thread Funktion `threadPoolFun()`. Die Funktionalität der Thread Funktion ist in Abschnitt 3.2.4 beschrieben.

3.2.2 Funktion: `void stopThreadPool(void)`

Die Funktion `void stopThreadPool(void)` stoppt die Ausführung der Threads. Da die Threads Messages aus der Queue lesen und damit blockieren bis neue Meldungen eintreffen, muss pro Thread eine Dummy Message gesendet werden um die Threads zu aktivieren. Die Dummy Meldung enthält dabei ein Task Objekt mit einer Task Funktion die ein `NULL` Pointer ist und so dem Thread mitteilt, dass er sich beenden soll.

Dieses einfache Verfahren stellt sicher, dass alle Nutz-Tasks in der Queue in Bearbeitung sind, bevor ein Thread die Meldung zum Beenden erhält.

Hinweis: Threads können auch mit einem Cancellation Verfahren beendet werden, dabei muss aber sichergestellt werden, dass alle Tasks fertig abgearbeitet sind, bevor die Threads cancelled werden werden können. In einer weiteren Aufgabe (Abschnitt 4) werden sie mit Hilfe einer zweiten Queue eine Funktion implementieren, die auf die Terminierung der gestarteten Threads warten kann.

3.2.3 Funktion: `void startTask(void *taskObject)`

Die Funktion `startTask(void *taskObject)` startet eine Task Funktion indem sie eine Meldung mit den beiden ersten Einträgen (this Pointer, Task Funktion) des Tasks Objekts an die Task Queue sendet.

Die Meldung dazu lässt sich einfach mit Casts erzeugen, ohne dass Daten kopiert werden müssen:

```
mq_send(taskQueue, (const char *)taskObject, sizeof(task_t), 0);
```

Der Typ `task_t` beschreibt ein minimles Task Objekt und ist in `threadpool.c` definiert. Das Task Objekt selbst muss auf den Typ des Buffers `(const char *)` gecastet werden.

Hinweis: die Funktion `mq_send()` blockiert wenn die Queue voll ist, d.h. der Zugriff auf die Task Queue ist implizit synchronisiert (man muss sich sich nicht um den Zugriff kümmern). Allerdings wird damit auch das aufrufende Programm blockiert und es können keine neuen Tasks gestartet werden, bis ein Platz in der Queue frei wird.

3.2.4 Thread Funktion: void *threadPoolFun(void *arg)

Die Thread Funktion öffnet zuerst die Task Queue und liest dann in einer unendlichen `while`-Schleife Meldungen aus der Queue. Die Meldungen müssen auf den Typ `task_t` gecastet werden und erlauben so Zugriff auf den `this` Pointer und die Task Funktion. Falls das Feld `func` dieser Struktur kein NULL-Pointer ist, muss die Funktion mit dem `this` Feld der Struktur als Parameter aufgerufen werden, andernfalls muss die while-Schleife mit `break` verlassen und der Thread terminiert werden.

3.3 Aufgabe 2

Implementieren und testen sie die drei Thread Pool Funktionen und die Thread Funktion. Dazu ist im Hauptprogramm ein einfacher Test implementiert, bei dem die Anzahl Threads und Tasks als Parameter übergeben werden können.

Die Tasks erhalten beim Start eine Task ID (Feld `tid` des Task Objekts) und legen diese als double Wert im Feld `retVal` des Task Objekt ab. Sind alle Task abgearbeitet, werden die `retVal`'s zusammengezählt und mit dem korrekten bzw. erwarteten Resultat verglichen. Arbeitet ihre Implementation korrekt?

4 Erweiterter Thread Pool

Der implementierte Thread Pool erlaubt es nicht, auf die Terminierung aller Tasks zu warten und dann eine neue Gruppe von Tasks zu starten. Zudem kann Thread Cancellation, wie oben beschrieben, nicht verwendet werden.

In dieser Aufgabe soll die Funktion `waitTasks()` implementiert werden, die solange wartet, bis alle bisher gestarteten Tasks beendet sind: die Funktion implementiert damit eine sogenannte Barrier.

4.1 Realisierung

Wir benötigen für die Realisierung eine zusätzliche Queue, die Wait Queue. Diese Queue verwenden wir aber nicht zur Übertragung von Information sondern zu Synchronisationszwecken. Dabei nutzen wird die Tatsache, dass das Lesen von einer leeren Queue blockiert.

- nachdem ein Task terminiert hat, sendet der dazugehörige Thread eine Dummy Meldung der Länge 1 an die Wait Queue
- die Funktion `waitTasks()` liest so viele Meldungen aus der Wait Queue wie bisher Tasks gestartet wurden
- die Anzahl gestarteter Tasks kann in der Funktion `runTask()` gezählt werden, verwenden sie dazu eine globale Variable mit Attribut `static`

4.2 Aufgabe 3

Kopieren sie ihr File `threadpool.c` aus Aufgabe 2 ins Verzeichnis "`a3`" und erweitern sie den Thread Pool um die oben beschriebene Funktionalität und implementieren sie `waitTasks()`.

Das Hauptprogramm enthält den gleichen Test wie bei Aufgabe 2, jedoch wird er hier zweimal hintereinander ausgeführt, wobei dazwischen ein `waitTasks()` eingefügt ist.

Starten sie das Programm wie folgt: `main.e 4 40`. Was geschieht? Wenn das Programm nicht hängen bleibt, haben sie erkannt, dass obige Beschreibung der Funktionalität zu einem Deadlock führen kann oder ihre Implementation ist noch fehlerhaft.

4.3 Aufgabe 4

Weil die maximale Anzahl Meldungen in einer Queue relativ klein ist (zehn) und das Senden von Meldungen blockiert, wenn die Queue voll ist, kann es in Aufgabe 3 zu einem Deadlock kommen:

- jeder Thread meldet der Wait Queue, wenn ein Task terminiert hat
- ist die Wait Queue voll, blockieren die Threads
- nun beginnt sich die Task Queue zu füllen, weil keine Task durch die blockierten Threads entnommen werden können
- ist die Task Queue voll, blockiert `mq_send()` in der Funktion `startTask()`
- wenn `startTask()` blockiert, kann ein nachgeschaltetes `waitTasks()` nicht ausgeführt werden und damit die Wait Queue nicht *entleeren*

Als Lösung kann man die Queues als nicht blockierend konfigurieren, die Handhabung ist in diesem Fall aber aufwendig.

Der Deadlock hat im Wesentlichen seine Ursache darin, dass die Wait Queue blockiert. Sorgt man nun dafür, dass nicht mehr Tasks gestartet werden als Messages von der Task Queue aufgenommen werden können, lässt sich ein Deadlock vermieden.

Die Lösung ist in diesem Fall einfach: die Funktion `startTask()` kennt die Anzahl Tasks die gestartet wurden und so viele Meldungen kann es im Maximum in der Wait Queue haben. Bevor ein Task gestartet wird, muss deshalb überprüft werden, ob die Anzahl gestarteter Tasks die Kapazitätsgrenze der Wait Queue übersteigen würde (Konstante `MYQ_MESGS`). Trifft das zu, muss zuerst eine Meldung aus der Wait Queue gelesen (bzw. entfernt) werden, sowie der Task Zähler nachgeführt werden. Wieso funktioniert das zusammen mit der Funktion `waitTasks()`?

Implementieren Sie das vorgeschlagene Verfahren. Funktionierts?

4.4 Aufgabe 5

Die Funktion `waitTasks()` kann in der Funktion `stopTaskQueue()` aufgerufen werden, um zuerst auf die Beendigung aller gestarteten Task zu warten. Damit wird es möglich mit Thread Cancellation die Threads zu stoppen.

Implementieren sie mit Hilfe der Funktionen `pthread_cancel()`, `pthread_setcancelstate()` und `pthread_setcanceltype()` die Terminierung des Threadspools. Welcher Cancel Typ soll verwendet werden und was geschieht mit den Threads die in der Queue blockieren?