Praktikum IPCPool

Task Queues und Thread Pools

Frühlingssemester 2020

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`. In `msgDefs.h` sind die Strings für die Namen der Queues definiert sowie die nowendigen Flags (`MYQ_FLAGS`), die maximale Anzahl Messages in der Queue (`MYQ_MESGS`) und die maximale Message Länge (`MYQ_MSGSZ`).

2.1 Aufgabe 1

Implementieren sie das Hauptprogramm in `main.c` und den Kind Prozess in `child.c`. Gehen sie dabei wie folgt 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, Achtung Pointer auf `attr` übergeben: `&attr`).

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_receive()`.

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

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

2.2 Terminierung Child

Im Kind-Prozess werden die Messages innerhalb einer `while(1)`-Schleife empfangen und gesendet. Um diese Schliefe verlassen zu können, erhält das Kind 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

Im folgenden werden wir einen einfachen Thread Pool mit einer Task Queue implementieren, dabei wird die Task Queue mit Hilfe einer POSIX Message Queue realisert. Die Threads des Pools holen sich dazu 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 Hilfe einer Funktion definiert.

3.1.1 Das Task Objekt

Um die Handhabung der Tasks zu erleichtern, arbeiten wir mit Task Objekten. Ein Task Objekt ist eine Struktur, die mindestens zwei Einträge hat und mit Hilfe folgender Typendefinition (`typedef`) beschrieben wird:

```
typdef struct taskObjectFoo {
    void *this;                     // mandatory
    void (*fun)(void *);            // madatory
     ....                           // user defined
} 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 Message in die Task Queue eingereiht werden .

Hinweis: für verschiedene Tasks müssen 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`).

Der Thread Pool besteht aus drei Funktionen, die in `threadpool.h` definiert sind:

- `void startThreadPool(int nThreads)`
- `void stopThreadPool(void)`
- `void runTask(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 das Attribut `static` für globale Variablen.

3.2.1 Funktion: `void startThreadPool(int nThreads)`

Die Funktion `startThreadPool()` 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 *Thread Cancellation* beendet werden, dabei muss aber sichergestellt werden, dass alle Tasks fertig abgearbeitet sind, bevor die Threads "cancelled" werden können. In einer weiteren Aufgabe (Abschnitt 4) werden sie diese Funktion implementieren.

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

Die Funktion `runTask(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 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 die Task Queue und liest dann in einer unendlichen `while`-Schleife Meldungen bzw. Task Objeket aus der Queue. Die Meldungen müssen auf den Typ `task_t` gecastet werden und erlauben so Zugriff auf den `this` Pointer (d.h. die Daten) und die Task Funktion. Ist das Feld `func` dieser Struktur kein NULL-Pointer, muss die Funktion `func` 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 erwarteten Resultat verglichen.

4 Erweiterter Thread Pool

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

In dieser Aufgabe soll die Funktion `waitTasks()` implementiert werden, die wartet, bis alle bisher gestarteten Tasks beendet sind. Eine Barrier in der üblichen Form kann hier nicht eingesetzt werden, weil beim Initialisieren der Barrier die Anzahl involvierter Tasks nicht unbedingt bekannt ist.

4.1 Realisierung

Für die Realisierung von `waitTasks()` benötigen wir einen globalen Zähler, der die Anzahl gestarteter Task erfasst, sowie eine Semaphore um `waitTasks()` zu blockieren bis sich jeweils ein Task beendet hat:

- die Anzahl gestarteter Tasks wird in der Funktion `runTask()` gezählt, verwenden sie dazu eine globale Zählvariable mit Attribut `static`
- jeder Task der terminiert,gibt die Semaphore mit `sem_post()` frei
- `waitTasks()` wartet in einer Schleife an der Semaphore mit `sem_wait()` bis bis jeweils ein Task termininiert hat und dekrementiert dann den globalen Zähler. Ist der Zähler auf `0` abgelaufen (alle gestarteten Tasks haben sich beendet), kann die Schleife verlassen werden.

Wichtiger Hinweis:
der Zugriff auf den globalen Zähler ist *kritisch*, da sowohl `runTask()` als auch `waitTasks()` aus parallel arbeitenden Threads aufgerufen werden können. Der Zugriff auf den Zähler muss also mit einem Mutex implementiert werden, verwenden sie dazu eine zweite Semaphore. Beachten sie weiter, dass das Dekrementieren des Zählers und der Test auf `0`, atomar sein muss (lokale Hilfsvariable verwenden).

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, 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.

4.3 Aufgabe 4

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.

Kopieren sie zuerst das File `threadpool.c` aus Aufgabe 3 ins Verzeichnis "a4".

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?