Sebastian Funck

10.07.2014

1 Einleitung

Die **GLib** ist ein quelloffenes und plattformunabhängiges Projekt, welches als Abhängigkeit lediglich die C-Standard Implementation benötigt. Dies ermöglicht es Software-Entwickler auf der ganzen Welt die **GLib** in ihre Projekten einzubinden und zu benutzten.

Die Entwicklung der **GLib** wird von der großen Community ***Das GNOME-Projekt*** vorrangetrieben, und jeder der Lust und Zeit hat darf mitwirken.

Das GNOME-Projekt betreut auch weitere Anwendungen und Projekte: **GIMP**, **GNOME-Desktop** und das ***GTK+-Framework***, um nur die großen und bekanntesten Namen zu nennen. Hieran kann man erkennen dass selbst größere Projekte auf die ***GLib*** zurückgreifen und diese sich bereits als zuverlässige und umfangreiche Bibliothek etabliert hat.

Bevor diese aus dem ***GTK+-Framework*** ausgegliedert und zusammen gefasst wurde, war die **GLib** in 5 Unterbibliotheken unterteilt. Diese Unterteilung kann man auch heute für eine sinnvolle Kategorisierung der Funktionalitäten nutzten:

• glib • GObject • GThread • GIO • GModule

Die Unterbibliothek ***Glib*** enthält sehr allgemeine Funktionen, diverse Datenstrukturen von einfachen Listen, über ausbalancierte Binärbäume bis hin zu einem kompletten Test-Framework, Stringformatierung und vieles mehr.

GObject ist ein komplettes Objektsystem für C um auch in C objektorientiert programmieren zu können. GObject ist gleichwohl wesentlich komplizierter als C++ und andere Hochsprachen, da man z.B. für eine simple Klasse die nur einen Konstruktor besitzt, schon mehr als 30 Codezeilen benötigt. Es ist jedoch eine hervorragende Schnittstelle auf der man aufbauen kann, so zum Beispiel basiert die Programmiersprache ***Vala*** auf ***GObject***.

Wie der Name schon verrät, stellt ***GThread*** Threading-Funktionalitäten zur Verfügung und erleichtert den Umgang mit Threads enorm.

Die ***GIO*** Unterbibliothek umfasst eine Reihe von Ein-/Ausgabefunktionen und vereinfacht so z.B. das Networking und Filehandling.

GModule ist ein Pluginsystem, mit dem man zur Laufzeit neue Funktionen zu einer Anwendung hinzufügen kann.

Ich werde im folgenden nur auf die **glib** eingehen und selbst hier nicht alles abdecken können, da dies den Rahmen sprengen würde.

2 glib

2.1 Primitive Datentypen

Da die **GLib** den Anspruch hat plattformunabhängig zu sein, liegt es nahe primitive Typen bereitzustellen die auf jeder Architektur durch gleich viele Bits dargestellt werden und den gleichen Wertebereich repräsentieren. Allerdings trifft dies nicht auf alle von der **GLib** eingeführte primitive Typen zu.

(signed)	(unsigned)	Speicher	typedef für	Bemerkung
`gboolean`		8 Bits	gint	TRUE, FALSE
`gint8`	`guint8`	8 Bits	signed char / unsigned char	
`gint16`	`guint16`	16 Bits	signed short / unsigned short	
`gint32`	`guint32`	32 Bits	signed int / unsigned int	
`gint64`	`guint64`	64 Bits	[2]	
`goffset`		64 Bits	gint64	Code Lesbarkeit
`gssize`	`gsize`	[1]	gint	Code Lesbarkeit
`gchar`	`guchar`	[1]	char / unsigned char	Vollständigkeitshalber
`gint`	`guint`	[1]	int / unsigned int	Vollständigkeitshalber
`gshort`	`gushort`	[1]	short / unsigned short	Vollständigkeitshalber
`glong`	`gulong`	[1]	long / unsigned long	Vollständigkeitshalber
`gfloat`		[1]	float	Vollständigkeitshalber
`gdouble`		[1]	double	Vollständigkeitshalber
`gpointer`		[1]	void*	
`gconstpointer`		[1]	const void *	

[1] Nicht zugesicherte Größe, [2] kein typischer typedef, *Abbildung 2.0*

Wie aus der *Abbildung 2.1* hervorgeht sind einige Typen lediglich der Vollständigkeithalber eingeführt worden oder sollen nur der Lesbarkeit dienen.

Auch neu sind die zwei typenlosen Zeiger `gpointer` und `gconstpointer`, die innerhalb der **GLib** fast ausschließlich verwendet werden. Auch wichtig zu erwähnen ist der Wert *NULL* der Zeigern zugewiesen werden kann und somit der Zeiger nun einen Nullzeiger repräsentiert.

2.2 Einfache Datenstrukturen

Darüber hinaus gibt es auch eine handvoll einfacher Datenstrukturen die wir als nächstes behandeln wollen.

2.2.1 Double/Single Linked List

GList - Die **GList** ist die doppelt-verknüpfte und die **GSList** die einfach-verknüpfte Variante einer Liste. Im folgenden werde ich nur auf die **GList** eingehen, analog lässt sich folgendes auch für die **GSList** sagen, mit der Ausnahme dass ein Element nicht seinen Vorgänger kennt.

Eine GList Variable ist in Wirklichkeit nur ein Element der Liste, deshalb sollte man vor allem bei **GSList** immer das vorderste Elemente der Liste irgendwo in der Hinterhand behalten. Dieses Element enthält einen Zeiger der auf das nächste Element, das vorherige Element und auf den Wert zeigt. Dies ist noch einmal in *Abbildung 2.1* verdeutlicht.

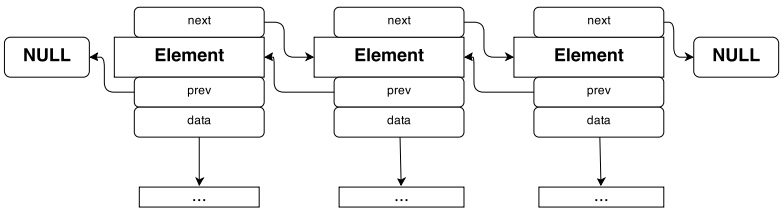

Abbildung 2.1: Doppelt verknüpfte Liste

Eine List erstellt man recht einfach in dem man einen **GList** Zeiger dem Wert *NULL* zuweist. Ein **GList** Zeiger der ein Nullzeiger ist repräsentiert die leere Liste und ist ein gültiges Argument für alle folgenden Listenfunktionen, allerdings sollte man sich davor hüten auf die Felder *next*, *prev* oder *data* zuzugreifen, da dies als Folge natürlich einen Fehler erzeugt.

Ein kleines Beispiel:

```
1 GList liste* = NULL;
2
3 liste = g_list_append(liste, "Erstes Element");
4 liste = g_list_append(liste, "Zweites Element");
5 liste = g_list_append(liste, "Drittes Element");
6
7 for (GList *elem = liste; elem != NULL; elem = li->next) {
8         g_printf("%s\n", li->data);
9 }
```

Ausgabe:

```
Erstes Element
Zweites Element
Drittes Element
```

In Zeile 1 initialisieren wir uns quasi eine leere Liste. Danach werden in Zeile 3-5 Elemente der Liste hinzugefügt und anschließend einmal über die Liste iteriert und die jeweiligen Zeichenketten Elemente ausgegeben.

Anhand diesem Beispiel kann man die Benutzung von Listen sehr einfach einsehen. Funktionen die auf Listen arbeiten haben immer einen Parameter in dem man die aktuelle Liste übergibt (in Wirklichkeit nur das Kopfelement der Liste) und nach Ausführung wird die bearbeitete Liste (das Kopfelement der bearbeiteten Liste) zurückgegeben.

Eine kleine Referenz mit den wichtigsten Funktionen:

`guint g_list_length(GList *list)`
Gibt die Anzahl der Elemente der Liste zurück.

`GList * g_list_append(GList *list, gpointer data)`
Hängt ein Element `data` an das Ende der Liste.

`GList* g_list_concat(GList *list1, GList *list2)`
Hängt die Liste `list2` and die Liste `list1` an.

`GList* g_list_insert(GList *list, gpointer data, gint position)`
Fügt ein Element an einer bestimmten `position` in der Liste ein.

`GList* g_list_remove(GList *list,gconstpointer data)`
Entfernt das erste Vorkommen von `data`.

`GList* g_list_remove_all(GList *list, gconstpointer data)`
Entfernt alle Vorkommen von `data`.

`GList * g_list_nth(GList *list, guint n)`
Gibt das n'te Element der Liste aus. Siehe auch `g_list_first(GList)` und `g_list_last(GList)`.

Für eine ausführliche Auflistung aller Funktionen sei hier auf die Gnome Developer Reference für Listen verwiesen
https://developer.gnome.org/glib/stable/glib-Doubly-Linked-Lists.html

2.2.2 Queues

Queues sind in **GLib** Reihen auf deren beiden Enden man übliche Stackfunktionen wie `pop`, `push` und `peek` andwenden kann. Es gibt für jede dieser drei Stackoperationen drei verschiedene Funktionen, einmal um auf den Kopf (`head`), dem Ende `tail` oder einem

beliebigen Element (`nth`) zuzugreifen.

Man hat daher die komplette Kontrolle über die Queue und kann auf alle Elemente zugreifen. Je nachdem was mit der Queue gemacht werden soll muss man sich auf bestimmte Funktionen beschränken. Will man etwa einen LIFO Stack wird man nur die Funktionen `g_queue_push_head`, `g_queue_pop_head` und `g_queue_peek_head` benutzen.

Für eine FIFO Queue zum Beispiel benötigt man nur folgende Funktionen: `g_queue_push_head`, `g_queue_pop_tail` und `g_queue_peek_tail`.

Ein LIFO-Stack Beispiel:

```
1  GQueue stack* = g_queue_new();
2
3  stack = g_queue_push_head ("Erstes␣Element");
4  stack = g_queue_push_head ("Zweites␣Element");
5  stack = g_queue_push_head ("Drittes␣Element");
6
7  for (gpointer elem = g_queue_pop_head(stack);
8      !g_queue_is_empty(stack);
9      elem = g_queue_pop_head(stack)) {
10
11         g_printf("%s\n", li->data);
12 }
```

Ausgabe:

```
Drittes Element
Zweites Element
Erstes Element
```

In der ersten Zeile initialisieren wir uns eine leere Queue, die wir als Stack benutzen wollen. Anschließend werden drei Elemente auf den Stack geschoben. In der for-Schleife wird solange bis der Stack keine Elemente mehr beinhaltet, ein Element nach dem anderen vom Stack genommen und ausgegeben.

Abschließend eine kleine Referenz der wichtigsten Queue Funktionen:

`GQueue * g_queue_new()`
Erzeugt eine leere Queue.

`guint g_queue_get_length (GQueue *queue)`
Gibt die Anzahl der Elemente in der Queue zurück.

`void g_queue_push_head (GQueue *queue, gpointer data)`
Fügt ein Element an den Anfang der Queue an. Siehe analog dazu `g_queue_push_tail` und `g_queue_push_nth`.

`void g_queue_pop_head (GQueue *queue, gpointer data)`
Entfernt das erste Element der Queue und gibt dieses auch zurück. Siehe analog dazu `g_queue_pop_tail` und `g_queue_pop_nth`.

`void g_queue_peek_head (GQueue *queue, gpointer data)`
Gibt das erste Element zurück, ohne es zu entfernen. Siehe analog dazu `g_queue_peek_tail` und `g_queue_peek_nth`.

`gboolean g_queue_is_empty (GQueue *queue)`
Gibt zurück ob diese Queue leer ist.

Für eine ausführliche Auflistung aller Funktionen sei hier auf die Gnome Developer Reference für Queues verwiesen
https://developer.gnome.org/glib/stable/glib-Double-ended-Queues.html

2.2.3 Weitere einfache Datenstrukturen

Es gibt durchaus noch mehr Datenstrukturen wie den `GHashTable`, eine Relation von Schlüsselelementen auf Werteelemente, den man als Relation (Map) oder als Menge (Set) benutzen kann.

Darüber hinaus gibt es noch ausbalancierte Bäume und Quarks (birelationale Abbildung von einem eindeutigen Integer auf eine Zeichenkette) und Sequenzen, die Listen mit einer Sortierdauer von $O(n\,log(n))$ sind.

3 Komplexere Strukturen

3.1 GFunc's

* Die **GLib** führt über Datenstrukturen und primitive Datentypen auch Funktionstypen ein. Diese sind normale Funktionszeiger, die jedoch anhand ihrer Benennung in der Funktionssignatur und der Parameteranzahl und Parametertypen einer bestimmte Funktionsart zuzuordnen.

`Rückgabewert Funktionsbezeichnung ( Parameter )`
Beschreibung der Funktionsart.

`gboolean GEqualFunc (gconstpointer , gconstpointer )`
Eine Funktion die zwei Objekte/Werte auf Gleichheit überprüft.

`guint GHashFunc (gconstpointer )`
Eine Funktion die einen Hashwert für ein Objekt/Wert erstellt.

`gint GCompareFunc (gconstpointer , gconstpointer )`
Eine Funktion die zwei Objekte/Werte auf ihre Ordnung überprüft und das Verhältnis zurückgibt.

`gboolean GSourceFunc (gpointer )`
Eine Funktion die etwas ausführt und zu TRUE oder FALSE auswertet. Der `gpointer` dient hier lediglich als Möglichkeit Daten in die Funktion zu bekommen und kann auch NULL sein.

`gpointer GThreadFunc (gpointer )`
Eine Funktion die bei Threads eingesetzt wird um die Logik auszuführen. Der Rückgabewert und der Parameter vom Typ **gpointer** dienen hier als Möglichkeit Daten in die Funktion zu bringen und Daten aus der Funktion zurück zu bekommen. Beide können **NULL** sein.

3.2 GMainLoop

Die **GLib** entstammt dem GTK+Framework, einem GUI-Framework und daher ist es nicht verwunderlich dass es viele Funktionalitäten gibt die die Oberflächenprogrammierung unterstützen. Zu diesen Funktionalitäten gehört auch eine Ereignisschleife (Eventloop).

Diese wird in der **GLib** als Mainloop (Hauptschleife) bezeichnet. Der Name ist etwas irreführend, da man meinen könnte es gibt nur eine Hauptschleife. Dies ist jedoch nicht der Fall, es können beliebig viele gleichzeitig existieren, es ist jedoch in den meisten Fällen nicht notwendig mehr als eine zu benutzten.

Einer Eventloop kann man Eventquellen hinzufügen und wenn ein bestimmtes Event ausgelöst wird, wird eine entsprechende Funktion ausgeführt.

Mögliche Eventquellen:

- Timeout - in zeitlichen Intervallen ausgelöste Events
- Signals - selbst implementierte Eventauslöser
- 'File-Watch' - Dateibeobachtung
- 'Child-Watch' - Beobachtung von Kinderprozessen

Auch erwähnt werden sollten die Idle-Funktionen, die immer ausgeführt werden wenn gerade kein Event ausgelöst wurde.

Für das folgende Beispiel sei hier ein Verweis auf alle Funktionen zur GMainLoop gegeben:
https://developer.gnome.org/glib/stable/glib-The-Main-Event-Loop.html

Ein Beispiel:

```
1   int main() {
2       GMainLoop *mainloop = g_main_loop_new( NULL , FALSE );
3       g_timeout_add( 1000 , countdown, mainloop );
4       g_main_loop_run( mainloop );
5       g_main_loop_unref( mainloop );
6       return 0;
7   }
8
9   gboolean countdown( gpointer data ) {
10      static int counter = 5;
11      g_printf("%d...", counter);
12      if (counter <= 0) {
13          g_main_loop_quit( (GMainLoop*)data );
14          return FALSE;
15      }
16      counter--;
17      return TRUE;
18  }
```

Das Codebeispiel besteht aus zwei Funktionen. Zum einen aus der Main-Funktion und zum anderen aus einer **GSourceFunc** mit der Bezeichnung `countdown`. Die Funktion `countdown` enthält hier unsere Logik die wir in der Mainloop ausführen wollen. EIn statischer Zähler (`counter`) wird auf 5 initialisiert und mit jedem Aufruf runtergezählt. Ist der Zähler bei 0 angekommen, wird die Mainloop beendet.

In der Main-Funktion erstellen wir uns in der zweiten Zeile eine neue **GMainLoop** die keinen Kontext besitzt (dies ist nur bei mehreren Mainloops notwendig) und erst von uns gestartet werden muss. Anschließend fügen wir unsere Funktion `countdown` einem Timeout-Event hinzu das alle 1000 Millisekunden ausgeführt wird und übergeben die Mainloop als Argument.

In der vierten Zeile wird nun die Mainloop ausgeführt. Solange die Mainloop läuft, liegt der Kontrollfluss in der Mainloop, d.h. die folgende Funktion `g_main_loop_unref` (dereferenziert die Mainloop) wird erst ausgeführt, sobald die Mainloop beendet wird und der Kontrollfluss abermals in der Main-Funktion liegt.

Ausgabe:

```
5... 4... 3... 2... 1... 0...
```

3.3 Zeichenketten

Kommen wir zu einem Thema dass in der C-Standard Implementation oftmals sehr umständlich ist. Die Sprache ist von dem Arbeiten mit Zeichenketten. Oft muss man mit Zeichenketten arbeiten und muss dafür auf unflexible `char`-Arrays zurückgreifen. Darum besitzt die **GLib** die **GString**s und viele Funktionen um diese zu manipulieren.

Um einen neuen String zu erzeugen rufen wir die Funktion `g_string_new()` auf und übergeben ein `char`-Array. Anschließend können wir diesen **GString** auf viele Arten modifizieren. Alle **GString**'s sind automatisch nullterminiert.

Das Gerüst des GString:

```
1    struct GString
2        gchar *str;
3        gsize len;
4        gsize allocated_len;
5    };
```

`GString * g_string_append (GString *string, const gchar *val)`
Fügt eine Zeichenkette an den **GString** an. Will man zwei **GString**s aneinanderfügen, so übergibt man als ersten Parameter den ersten **GString** und als zweiten Parameter `*str` des zweiten **GString**.

`GString * g_string_prepend (GString *string, const gchar *val)`
Fügt eine Zeichenkette vor den **GString** an.

`GString * g_string_insert (GString *string, gssize pos, const gchar *val)`
Fügt eine Zeichenkette in einen **GString** an einer bestimmten Position ein. Der initiale **GString** wird nicht abgeschnitten sondern erweitert.

Für eine ausführliche Auflistung aller Funktionen sei hier auf die Gnome Developer Reference für Strings verwiesen
https://developer.gnome.org/glib/stable/glib-Strings.html

3.3.1 Kodierung

Die **GLib** arbeitet intern ausschließlich mit dem UTF-8 Format und auch **GString**s liegen im UTF-8 Format vor. Es ist jedoch von Zeit zu Zeit notwendig das Zeichenketten in andere Kodierungen umgewandelt werden für zum Beispiel externe Anwendungen oder Bibliotheken.

Es gibt neben dem `gchar` (UTF-8) auch noch `gunicode2` (UTF-16) und `gunicode` (UTF-32) Zeichen. Zeichenketten im UTF-16 und UTF-32 Format sind somit nur als Arrays verfügbar.

Es gibt eine sehr mächtige Konvertierungsfunktion `convert` die beliebig zwischen den verfügbaren Kodierungen konvertieren kann (es werden auch weitere Kodierungen außer UTF unterstützt).

Ein Beispiel:

```
1 GString* str_utf8;
2 gunichar2* str_utf16;
3
4 str_utf8  = g_string_new("Ich␣bin␣ein␣String!\n");
5 str_utf16 = g_convert(str_utf8->str, -1,
6                        "UTF-16BE", "UTF-8",
7                        NULL, NULL, NULL );
```

Dieses Beispiel konvertiert eine UTF-8 Zeichenkette in eine UTF-16 Zeichenkette. Die drei NULL Argumente kann man in der Regel vernachlässigen da hierdurch man informiert werden kann wie viele Bytes gelesen und geschrieben wurden.

3.3.2 String Utility Funktionen

Es sei noch kurz erwähnt das es unter den String Utility Funktionen weitere Funktionen gibt, die dem Themengebiet Zeichenketten zuzuordnen sind.
https://developer.gnome.org/glib/stable/glib-String-Utility-Functions.html

3.4 Threads

3.4.1 Allgemeines

Threads sind Konstrukte die benutzt werden um Programmcode parallel auszuführen.

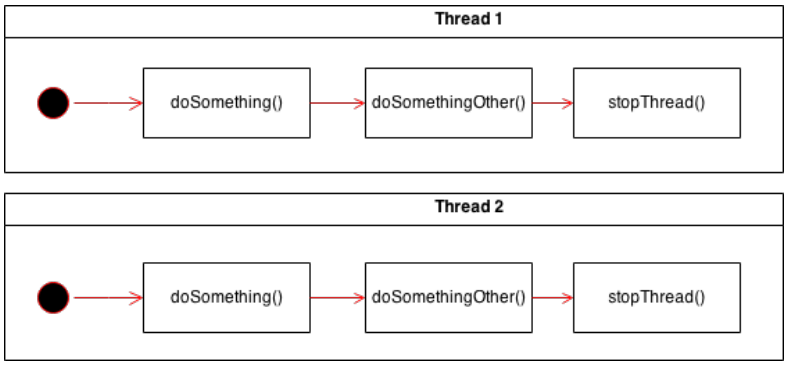

Abbildung 3.1:

Selbstverständlich stellt die **GLib** auch Funktionen zur Thematik Threads zur Verfügung, die die Benutzung von diesen vereinfachen soll. Ohne viel Aufwand lässt sich ein solcher Thread schon erzeugen. Ein Beispiel:

```
1   int main() {
2       int *value1;
3       int *value2;
4
5       *value1 = 10000;
6       *value2 = 20000;
7
8       g_printf( "Starte␣Thread␣1␣..." );
9       GThread *thread1 = g_thread_new("Thread␣1", thread_func, value1);
10
11      g_printf( "Starte␣Thread␣2␣..." );
12      GThread *thread2 = g_thread_new("Thread␣2", thread_func, value2);
13
14      g_printf( "Warten␣auf␣Thread␣2␣..." );
15      g_thread_join( thread2 );
16
17      g_printf( "Warten␣auf␣Thread␣1␣..." );
18      g_thread_join( thread1 );
19
20      return 0;
21  }
22
23  gpointer thread_func( gpointer data ) {
24      int* counter = (int*) data;
25      while ( counter > 0 ) {
26                      g_printf("%d...", counter);
27      }
28      g_thread_exit( NULL );
29      return NULL;
30  }
```

Für eine ausührliche Dokumentation der Thread-Funktionen sei hier auf die Gnome Developer Reference verwiesen: https://developer.gnome.org/glib/stable/glib-Threads.html

Alles was dieses Codebeispiel macht ist zwei Thread zu erstellen, die parallel zueinander zwei Zähler herunterzählen. Dies ist ein sehr einfaches Beispiel, da Threads heutzutage jedoch immer mehr an Relevanz gewinnen, sollte man dieses Gebiet nicht unterschätzen. Ich werde im weiteren nicht näher auf Anwendungsgebiete oder Beispiele von Threads eingehen, sondern Konzepte der **GLib** zum Thema Threads vorstellen.

3.4.2 AsyncQueue

Bei der Thematik des parallelen Rechnens wird man einmal an den Punkt gelangen an dem zwei oder mehrere Threads mit einander kommunizieren müssen. Da Threads eines Prozess den gleichen Speicher teilen könnte man leicht auf die Idee kommen einfach in einen festen Bereich Daten reinzuschreiben und mit anderen Threads auszulesen. Dies ist allerdings ein Problem, da mehrere Threads versuchen könnten gleichzeitig zu lesen oder zuschreiben. Dies führt oft zu unerwarteten Problemen und macht eine Fehlersuche sehr kompliziert bis unmöglich.

Für diese Problematik gibt es die asynchrone Queue. Mithilfe dieser kann man gleichzeitig Elemente zur Queue hinzufügen und von ihr Lesen, ohne dass sich Threads in die Quere kommen. *Abbildung 3.2* verdeutlicht dieses Konzept.

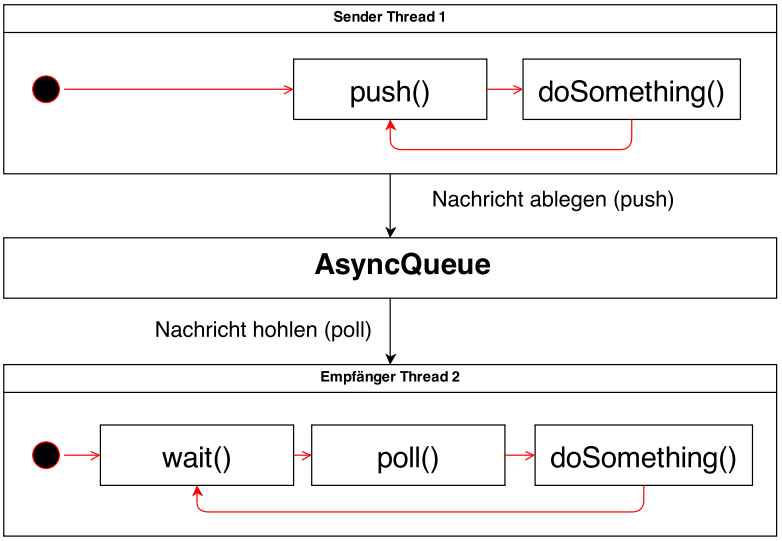

Abbildung 3.2: Asynchrone Queue

3.4.3 Threadpool

Ein weiteres Konzept ist der Threadpool. Benötigt man oft Threads für sehr kurze Berechnungen kann das Erzeugen des Thread einen großen Teil der Zeit einnehmen. Um die Neuerzeugung der Threads zu umgehen kann man sich einen Threadpool anlegen. Man spezifiziert wie viele Threads maximal in dem Pool zur Verfügung stehen sollen und sagt dem Threadpool anschließend nur dass man einen Thread benötigt. Der Threadpool kümmert sich hierbei um das Thread Management.
Wie ein Threadpool funktioniert wenn ein Thread benötigt wird, verdeutlicht *Abbildung 3.3*.

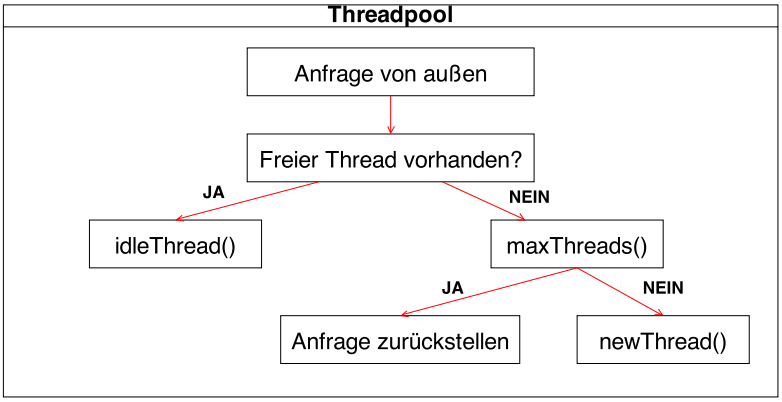

Abbildung 3.3: Threadpool

4 Zusammenfassung

Wie sie bestimmt bemerkt haben ist die **GLib** sehr umfangreich, wenn man betrachtet dass in dieser Ausarbeitung nur eine Unterbibliothek behandelt wurde und selbst an dieser wurde nur an der Oberfläche gekratzt. Der komplette Umfang der **GLib** würde sicherlich mehrere Bücher füllen. Auch das eine ganze Programmiersprache (Vala) auf einem Teil der **GLib** basiert verstärkt diesen Eindruck.

Mit der **GLib** und dem **GTK+-Framework** hat man ein sehr mächtiges Toolset mit dem man komplexe Anwendungen realisieren kann, aber auch schon in kleineren Anwendungen werden diese erfolgreich eingesetzt. Zudem hat sich die **GLib** in größeren Projekten bewährt wie unter anderem in dem **Symbian OS**.

5 Quellen

[1] Das GNOME Developer GLib Reference Manual: https://developer.gnome.org/glib/stable/
(Stand: 04.06.2014)

[2] E-Book: GTK+-Gnome-Application-Development:
http://book.huihoo.com/gtk+-gnome-application-development/ggad.html
(Stand: 04.06.2014)

[3] Wikipedia: GObject
http://de.wikipedia.org/wiki/GObject
(Stand: 04.06.2014)

[4] Nokia Symbian Guide:
http://docs.huihoo.com/symbian/nokia-symbian3-developers-library-v0.8/GUID-94D67092-5EB3-4D83-A164-CA628F2E2DB0.html
(Stand: 04.06.2014)