Concurrent Programming 2

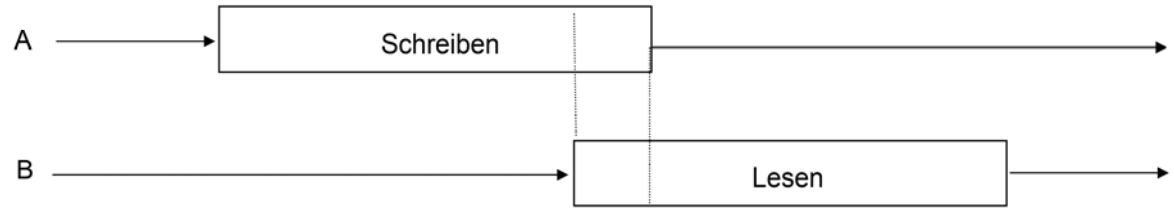
1 Monitor

1.1 Allgemein

1.1.1 Klasse Monitor

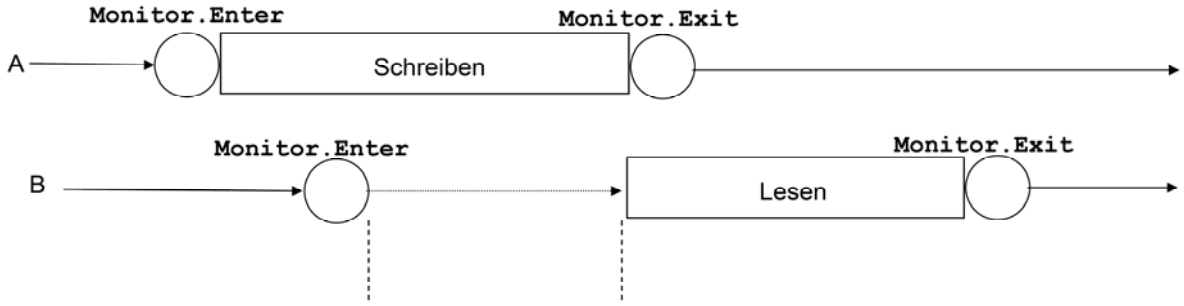
Diese Klasse verhindert, dass zu einem gegebenen Zeitpunkt mehr als ein Thread auf eine Ressource zugreifen kann.

Monitore helfen folgende Situation zu vermeiden:



Thread A schreibt in einen Speichebereich; Thread B liest aus diesem Bereich. Dabei können sich die beiden Vorgänge überschneiden. D.h. B liest, obgleich A noch nicht fertig ist.

Monitore synchronisieren Zugriff auf den Speicherbereich:



Thread B wird blockiert, bis Thread A Exit erhält! Jeder Thread wendet sich an den Monitor, wenn er auf das synchronisierte Objekt zugreifen möchte. Das Monitor-Objekt serialisiert den Zugriff auf das durch den Monitor überwachte Objekt. Dadurch erfolgen die Lese- und Schreibvorgänge nacheinander!

1.1.2 Synchronisationsobjekte

Das Synchronisationsobjekt ist wie ein "Wecker" (`Monitor.Enter(syncObj);`), mit dem ein oder mehrere Threads sich für eine bestimmte Zeit schlafen legen können (`Monitor.Wait(syncObj);`) und wenn ein Pulse (`Monitor.Pulse(syncObj);`) kommt wieder aufwachen.

1.1.3 C#-Code

```
//Den Monitor auf ein bestimmtes Synchronisationsobjekt einstellen
Monitor.Enter(syncObj);
```

```
//Lässt die Funktion deren Monitor das ist so lange warten bis sie von dem
Synchronisationsobjekt einen Pulse bekommt
Monitor.Wait(syncObj);

//Die Synchronisation auf ein bestimmtes Synchronisationobjekt beenden
Monitor.Exit(syncObj);

//Diese Funktion lässt das Synchronisationsobjekt einen Pulse schicken der
eine darauf synchronisierte Funktion erwachen lässt
Monitor.Pulse(syncObj);

//Mit diesem Befehl kann eine Funktion das Synchronisationsobjekt für sich
sperren, dies wird zum Beispiel verwendet wenn ein Datensatz von
verschiedenen Funktionen aufgerufen wird um gleichzeitigen Zugriff zu
vermeiden
lock (syncObj)
{
//Code
}
```

2 Ringbuffer

2.1 Allgemein

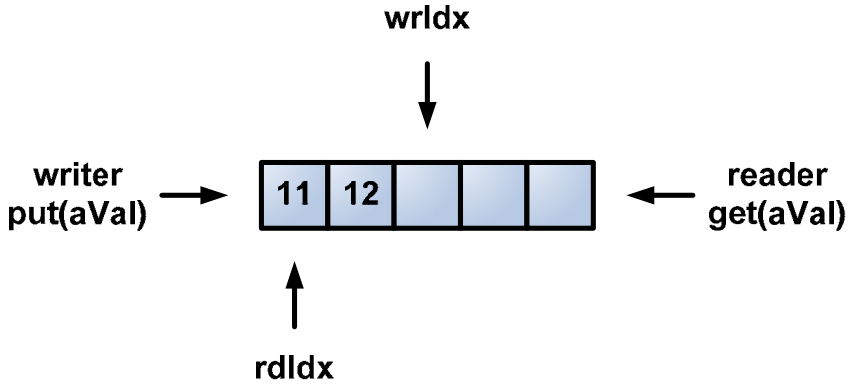
Die Verwendung eines Ringbuffers ist eine effektive Möglichkeit, Probleme des nebenläufigen Zugriffs zu umgehen: Es ist immer noch am besten, überhaupt keinen nebenläufigen Zugriff durchzuführen.

Der Ringbuffer verwendet einen Algorithmus namens „Producer und Consumer" - einer der Beteiligten schiebt Daten in den Buffer hinein, und der andere holt Daten heraus. Es gibt keinen nebenläufigen Zugriff, wenn es genau einen Producer und genau einen Consumer gibt.

Ein Ringbuffer wird über zwei Zeiger angesprochen: `writer` und `reader`. `writer` ist die Stelle, an der Daten geschrieben werden, und wird nur vom Producer verändert. Das Lesen der Daten geschieht von `reader`, was wiederum nur vom Consumer verändert wird.

Der Ringbuffer läuft so lange glatt, bis er voll ist. Wenn das passiert, wird es knifflig. Sie haben dann verschiedene Möglichkeiten: Wenn writer hinter `reader` läuft, dann ist ein voller Buffer an Daten verloren. Es besteht die Möglichkeit den Producer anzuhalten oder eines vorübergehenden zusätzlichen Buffers, der für den Hauptbuffer einspringt.

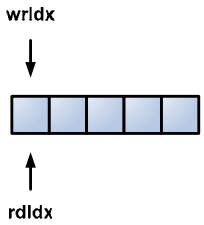
2.2 Skizze

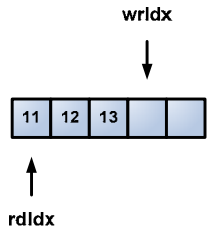


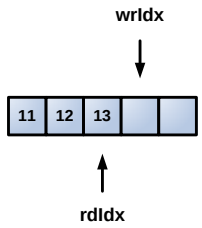
2.3 Bsp Ringbuffer

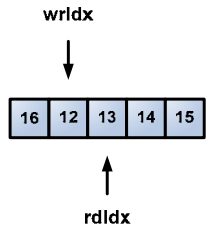
2.3.1 Programmablauf

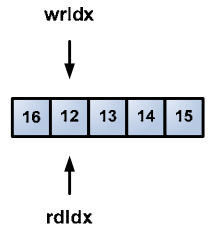
Nr.	Skizze	Befehl	Werte	Beschreibung

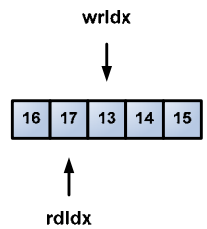
1.)			`itemCount = 0`
			`wrIdx = 0`
			`rdIdx = 0`
				Es befindet sich nichts im Ringbuffer. `wrIndx` und `rdIdx` stehen in ihrer Startposition.

2.)		`put(11);`
		`put(12);`
		`put(13);`
			`itemCount = 3`
			`wrIdx = 3`
			`rdIdx = 0`
				Der `writer` schreibt eine Variable via `put(aVal)` in den Buffer an der Stelle wo der `wrIdx` hinzeigt. `wrIdx` spring dann um einen Wert weiter.

3.)		`get(aVal); //11`
		`get(aVal); //12`
			`itemCount = 1`
			`wrIdx = 3`
			`rdIdx = 2`
				Der `reader` liest eine Variable via `get(aVal)` aus dem Buffer an der Stelle wo der `rdIdx` hinzeigt. `rdIdx` spring dann um einen Wert weiter.

4.)		`put(14);`
		`put(15);`
		`put(16);`
			`itemCount = 4`
			`wrIdx = 1`
			`rdIdx = 2`
				Hier überschreibt der `writer` einen schon beschriebenen Bereich im Ringbuffer. Dies macht aber nichts da dieser Wert schon ausgelesen wurde.

5.)		`get(aVal); //13`
		`get(aVal); //14`
		`get(aVal); //15`
		`get(aVal); //16`
			`itemCount = 0`
			`wrIdx = 1`
			`rdIdx = 1`
				Es werden die noch übrigen Werte vom `reader` ausgelesen. Es besteht wieder die Möglichkeit den gesamten Ringbuffer zu beschreiben.

6.)		`put(17);`	`itemCount = 1`
			`wrIdx = 2`
			`rdIdx = 1`
				Es wird wieder ein Wert mit dem `writer` in den Ringbuffer geschrieben. Es ist eine Stelle des Ringbuffers belegt (itemCount = 1).

2.3.2 C#-Code

Siehe Programmbeispiel: ProdCons1_Comment.cs

3 Das Producer Consumer Problem

3.1 Allgemein

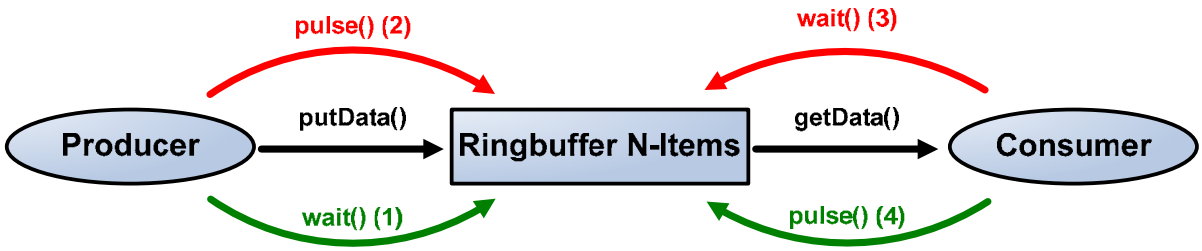
Zwei Prozesse benutzen einen gemeinsammen allgemeinen Puffer (Ringbuffer) mit fester Größe. Einer von ihnen, der Producer, legt Informationen in den RB, der andere, der Consumer, nimmt sie heraus.

Es können verschiedene Probleme auftreten:

- Ein Problem entsteht, wenn der Producer eine neue Nachricht in den RB legen möchte, der aber schon voll ist bzw. wenn der Consumer eine oder mehrere Nachricht aus dem RB heraus holen möchte, aber dieser leer ist (Lösung: Bsp Producer Consumer).
- Legt ein Producer gerade ein Element in den RB oder entfernt ein Consumer gerade ein Element, so muss verhindert werden, dass ein anderer Producer- oder Consumerprozess diesen Vorgang unterbricht, um auch auf den RB verändernd zuzugreifen. Andernfalls kann es zu einem inkonsistenten Zustand der Datenstruktur kommen.

3.2 Bsp Producer Consumer

3.2.1 Skizze



3.2.2 Beschreibung

Wenn etwas aus dem Synchronistationsobjekt Ringbuffer ausgelesen oder hinein geschrieben werden soll, wird als erstes überprüft ob dies möglich ist (`Monitor.Enter(rb);`). Sollte der RB voll sein wird der Producer so lange schlafen gelegt bis sich der Zustand ändert bzw. ihn der Consumer aufweckt (`Monitor.Wait(rb);`). Sollte der RB leer sein wird der Consumer schlafen gelegt bis sich der Zustand ändert bzw. ihn der Producer aufweckt.

Ist der RB weder voll noch leer kann er beschrieben oder ausgelesen werden. Allerdings ist nur einer der beiden Abläufe möglich. D.h. ist die Überprüfung des Befüllungszustandes des RB beendet (`Monitor.Exit(rb);`) hat jeweils der Producer oder der Consumer exklusiven Zugriff um Daten abzulegen bzw. zu kunsumieren (`lock (rb)`).

Beschreiben und auslesen eines RB siehe Bsp Ringbuffer.

Wenn der RB fertig beschrieben bzw. ausgelesen wurde und der RB nun voll (`wasFull = rb.isFull();`) bzw. leer war (`wasEmpty = rb.isEmpty();`) wird der

Producer bzw. der Consumer aufgeweckt (`Monitor.Pulse(rb);`). Zum Abschluss wird der aktuelle Thread für die angegebene Anzahl von Millisekunden blockiert (`Thread.Sleep(200);`).

(1) Tritt nur in Kraft, wenn der Producer auf einen vollen RB trifft.

(2) Tritt nur in Kraft, wenn der Producer auf einen leeren RB trifft (Consumer wird aufgweckt).

(3) Tritt nur in Kraft, wenn der Consumer auf einen leeren RB trifft.

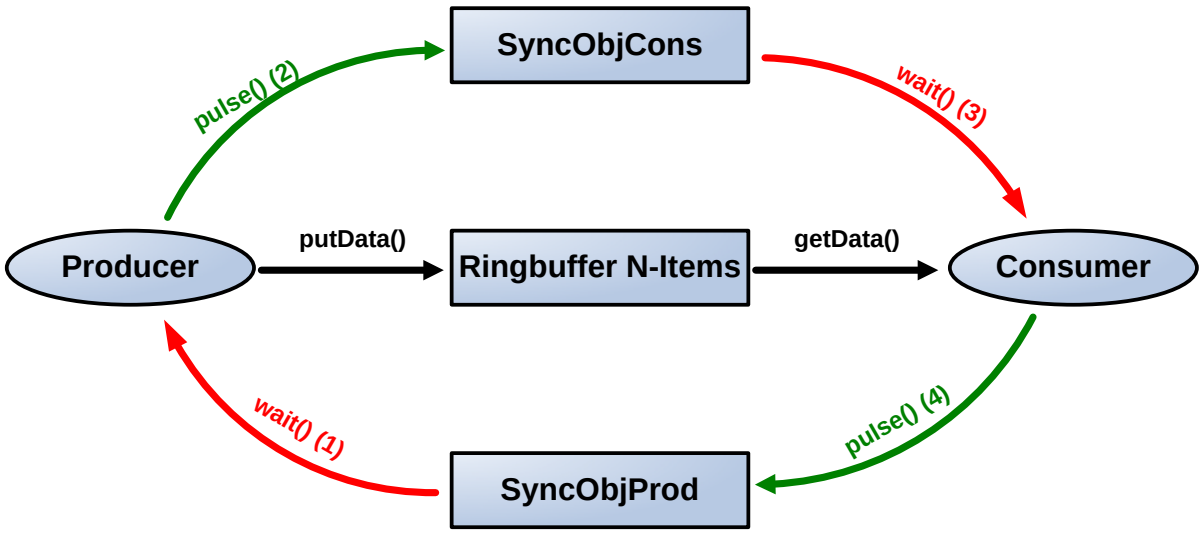
(4) Tritt nur in Kraft, wenn der Consumer auf einen vollen RB trifft (Producer wird aufgeweckt).

3.2.3 C#-Code

Siehe Programmbeispiel: ProdCons1_Comment.cs

3.3 Bps Producer Consumer 2

3.3.1 Skizze



3.3.2 Beschreibung

Hier handelt sich um eine erweiterte Form des normalen Producer Consumer Bsp. Im gegensatz zum Standartpsb. wird hier der RB entlastet in dem zwei zusätzliche SyncObjekte (SyncObjProd und SyncObjCons) für den Producer und den Consumer hinzugefügt wurden. Es sind jedoch die Grundfunktionen des Bsp Producer Consumer in diesem Bsp. enthalten. Außerdem wurde das auslesen und beschreiben des RB auch nicht verändert.

(1) Tritt nur in Kraft, wenn der Producer auf einen vollen RB trifft.

(2) Tritt nur in Kraft, wenn der Producer auf einen leeren RB trifft (Consumer wird aufgweckt).

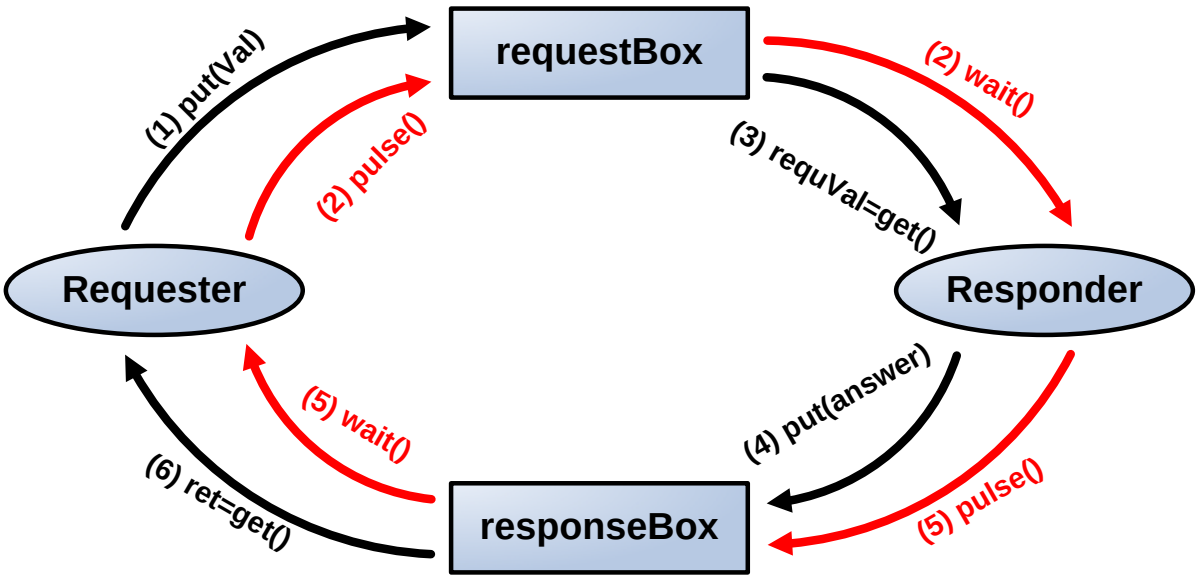
(3) Tritt nur in Kraft, wenn der Consumer auf einen leeren RB trifft.

(4) Tritt nur in Kraft, wenn der Consumer auf einen vollen RB trifft (Producer wird aufgeweckt).

4 Request Response

4.1 Bsp Request Reponse

4.1.1 Skizze



4.1.2 Beschreibung

Als erstes sendet der Requester an den Responder via Synchronistationsobjekt requestBox eine Anfrage in dem er das Synchobjekt sperrt (`lock (requestBox)`), eine Variable an die requestBox übergibt (`requestBox.put(val);`) und die requestBox aufweckt (`Monitor.Pulse(requestBox);`).

Der Responder wartet auf die Anfrage und richtet seine Aufmerksamkeit auf sein Synchronisationsokjekt (`Monitor.Enter(requestBox);`). Es wartet dann so lange bis ein Pulse von der requestBox kommt (`Monitor.Wait(requestBox);`). Sollte der Pulse kommen übernimmt der Reponder die Daten von seinem Synchronisationsokjekt (`requestVal = requestBox.get();`).

Die Daten werden dann verarbeitet. In unserm Bsp. wird die Verarbeitungszeit simuliert (`Thread.Sleep(1000);`).

Danach wird eine Antwort gesendet. Die responseBox wird gesperrt (`answer = requestVal + 2;`), es wird die Antwort hinein geschrieben (`responseBox.put(answer);`) und es wird ein Pulse an das Synchronisationsokjekt des Requesters gesendet (`Monitor.Pulse(responseBox);`).

Der Requester wartet auf seine Antwort in dem er seine Aufmerksamkeit auf die responseBox richtet (`Monitor.Enter(responseBox);`) und auf das aufwecken wartet (`Monitor.Wait(responseBox);`). Sollte der Pulse kommen übernimmt der Requester die Daten von seinem Synchronisationsokjekt (`ret = responseBox.get();`).

4.1.3 C#-Code

Siehe Programmbeispiel: RequestResponse1.cs