Externe Speicheralgorithmen: Stack und Queue auf SD-Karten mit der Wiselib

Dominik Krupke

February 15, 2013

1 Einleitung

Im Gegensatz zu den meisten anderen Datenstrukturen hat man bei der Stack und der Queue den Vorteil, dass man jeder Zeit weiss, welches Element als naechstes gelesen wird. Da man somit weiss, wo sich die gewuenschten Daten befinden, sind keine komplizierten Algorithmen zur Blocksuche noetig und die einzigen optimierbaren Prozesse sind die Schreibe und Lesevorgaenge. Aufgrund der vorhin genannten Schreibe und Leseordnung kommt hierbei nur ein Buffer infrage. Durch das spezielle Blockinterface und die fehlende dynamische Speicherverwaltung gibt es hierbei jedoch durchaus spezielle Designentscheidungen zu treffen, um am Ende eine moeglichst schnelle und persistente Datenstrukturen zu haben. Im Verlaufe des Algorithmik-Praktikums habe ich etliche Varianten implementiert. Die Grundideen die ich dabei entwickelt habe, sowie die letztenendliche Implementation moechte ich hier nun vorstellen.

2 Eigenschaften des externen Speichers

Die SD-Karte als externer Speicher unter der Wiselib hat einige Eigenschaften, die es bei der Implementierung der Datenstrukturen zu beachten gibt:

- Es koennen nur ganze Bloecke gelesen oder geschrieben werden. Diese haben eine Groesse von gewoehnlich 512Byte. Es koennen auch mehrere zusammenhaengende Bloecke sowohl gelesen als auch geschrieben werden. Der hierbei verwendete RAM-Block muss dabei natuerlich zusammenhaengend sein.
- Eine Leseoperation eines Blockes braucht etwa 1-2ms unabhaengig von der Anzahl der gleichzeitig angeforderten Bloecke.
- Eine Schreibeoperation eines Blockes dauert gewoehnlich einige ms. Es kommt auf den meisten SD-Karten zusaetzlich bei etwa jeder 127. Schreibeoperation zu einem Peak von ueber 200ms. Die durchschnittliche Zeit pro Block kann durch das Schreiben mehrerer Bloecke enorm gesenkt werden.

Anmerkung: Es wurde ausschliesslich mit Arduinos gearbeitet. Es ist aber anzunehmen, dass die anderen Wiselib-Mikrokontroller sich genauso verhalten.

3 Buffer

Durch die fehlende dynamische Speicherverwaltung muss auf statische Buffer zurueckgegriffen werden. Hierfuer gibt es 3 Varianten:

1. Einen Externen Buffer, der mit Kontainern (z.B. Arrays) arbeitet
2. Einen Itembasierten Internen Buffer, der eine bestimmte Anzahl von Elementen buffert.
3. Einem Blockbasierten Internen Buffer, der ganze Bloecke im Hauptspeicher haelt und mit dem externen Speicher austauscht.

Wir werden zwar gleich auf die zweite Variante am intensivesten eingehen, uns aber am Ende auf die dritte festlegen. Dies haengt damit zusammen, dass die zweite Variante die komplizierteste ist und es durchaus Faelle gibt, indem sie zu bevorzugen ist. Die dritte Variante hat im Vergleich zur 2. einen viel hoeheren Minimalbedarf im Hauptspeicher, ist dafuer aber im Gegenzug sehr schnell und recheneffizient, ein Faktor der bei Mikrokontrollern nicht zu vernachlaessigen ist.

3.1 Externe Buffer

Die Idee hinter diesem Buffer ist, dass wenn ein Anwender mehrere Elemente einfuegen oder auslesen will, diese gebuendelt uebergibt oder erhaelt. So koennte der Anwender, wenn er weiss, dass er die naechsten drei Elemente vom Stack fuer seine Berechnung braucht, diese nicht einzelnd anfordern, sondern gleich alle drei auf einmal. Auf diese Weise kann man sich eine Menge IOs auf den externen Speicher sparen. Sollten die Elemente gut im Hauptspeicher liegen, ergibt sich fuer eine Operation trotz mehrerer Elemente im Idealfall nur ein IO. Leider setzt diese Methode aber auch eine hohe Kooperationsbereitschaft und/oder Kompetenz des Anwenders vorraus. Es kann auch sein, dass sein Algorithmus ueberhaupt nicht die Moeglichkeit zum Buendeln laesst bzw. diese sehr erschwert. Aus diesem Grund ist es anwenderfreundlicher und sinnvoller das 'Buendeln' einfach der Datenstruktur zu ueberlassen, zumal das, wie wir sehen werden, moeglich ist. Der externe Buffer ist lediglich die einfachste Idee zur Optimierung und es kann durchaus Faelle geben, in der sich eine sinnvolle Variante implementieren lassen wuerde. Im Vergleich zu den anderen Varianten, liegt ihr Vorteil darin, dass Datentypen dieser Variante nahezu keinen Speicher brauchen und daher die ganze Zeit ueber auch in grosser Anzahl im Speicher gehalten werden koennen (Stellen Sie sich vor, sie brauchen 30 verschiedene Stacks. Die in diesem Praktikum entstandene, sehr effiziente, Version braucht aber mindestens 512Byte Speicher. Somit braeuchte man ganze 15kB RAM, die auf den meisten Mikrokontrollern aber nicht zur Verfuegung stehen. Fuer dieses Problem kann man aber durch die Persistenz der Datenstrukturen ein Workarround bauen.).
Es ist zu Beruecksichtigen, dass diese Variante zwar einen sehr niedrigen konstanten Speicherverbrauch hat, aber trotzdem fuer IOs auf den externen Speicher kurzfristig mindestens 512Byte zusaetzlichen Speicher braucht.

3.2 Interne Buffer

Bei internen Buffern, wird das Buendeln von Operationen der Datenstruktur ueberlassen. Dies funktioniert deutlich effizienter, als wenn der Anwender sich selber darum kuemmern wuerde. Es koennen naemlich nicht nur gleiche Operationen gebuendelt, sondern auch verschieden miteinander verrechnet werden. Z.B. kann bei einem Stack ein Push gefolgt von einem Pop ohne IO auf den externen Speicher ausgekommen werden. Mithilfe von internen Buffern lassen sich einige Optimierungen verwirklichen, die mit externen Buffer nicht moeglich waeren. Dafuer benoetigen sie aber fuer die Dauer ihrer Existenz zusaetzlichen Speicher.

3.2.1 Itembasierte interne Buffer

In dieser Buffervariante wird ein konstanter Buffer in Form eines Arrays ueber N Elementen in der Datenstruktur angelegt. Saemtliche Operationen laufen hierbei ueber den Buffer, welcher nach Bedarf gefuellt oder geleert wird. Sollte man ein Element einfuegen wollen, so wird versucht dieses in den Buffer zu kopieren. Sollte dieser bereits voll sein, so werden einige Elemente aus dem Buffer auf den externene Speicher ausgelagert und der Versuch wiederholt. Der Lesevorgang verlaeuft relativ analog: Sollte der Leseplatz leer sein, so werden einige Elemente aus dem externen Speicher in den Buffer kopiert.
Durch den Buffer ist es uns moeglich Elemente effizient zu sammeln und pro IO auf den externen Speicher moeglichst viele Elemente zu uebertragen. Es ist hierbei jedoch zu beachten, dass der Buffer im Hauptspeicher gewoehnlich nicht Bitkompatibel mit dem Blockinterface des externen Speichers ist (Ein Array basiert auf Elementen, die Daten im externen Speicher auf Bloecken). Wir werden daher fuer jeden IO einen temporaeren Block im RAM anlegen muessen und fuer den Auslagerungsvorgang die Elemente aus dem Buffer in diesen Block kopieren, um ihn dann anschliessend auf den externen Speicher zu uebertragen. Analoges gilt fuer den Einlagerungsvorgang. Der Geschwindigkeitsvorteil durch das Uebertragen von mehreren Bloecken am Stueck auf den externen Speicher laesst sich daher nur schwer ausnutzen. Wir koenne weder Testen wie viel Platz im Hauptspeicher noch frei ist, noch dynamisch Speicher reservieren. Wir muessten also im Vorraus bestimmen, wie viele Bloecke uebertragen werden sollen und diese dann fuer die Dauer der Operation im Hauptspeicher anlegen. Wir stehen also vor der Entscheidung zwischen Geschwindigkeit und RAM-Verbrauch. Das dies nicht so sein muss, werden wir beim Blockbasierten internen Buffer sehen.
Ein weiterer Punkt, der sich fast indirekt aus dem vorherigem Absatz ergibt, ist die Art der Blockausnutzung. Gewoehnlich ist von partiellem Schreiben abzusehen, da uns keine echte Funktion dafuer zur Verfuegung steht und wir somit, um Datenverlust zu vermeiden, diese Funktion aus einer Read und einer Write Operation modelieren muessten. Dies wuerde aber im WorstCase eine Verdopplung der IOs bedeuten. Es ist hierbei also zu empfehlen nur abgeschlossene Bloecke zu schreiben und somit automatisch auch nur zu lesen. Dabei ergeben sich wieder zwei Optionen:

1. Wir schreiben pro Block eine konstante Anzahl von Elementen, welche am Anfang durch die Groesse des Buffers moeglichst optimal berechnet wird. Diese Variante erschwert uns die Persistenz, fuer den Fall, dass

wir die Buffergroesse im Programmverlauf aendern wollen, weil uns dann mehr oder weniger Speicherplatz zur Verfuegung steht. Sollte der Buffer kleiner werden, so koennte es passieren, dass die Bloecke nichtmehr vollstaendig in den Buffer geladen werden koennten und wir wieder partielles Lesen und Schreiben benutzen muessten, was aber zusaetzlichen Verwaltungsaufwand bedeuten wuerde (Da wir dann ploetzlich einen Block anderer Groesse haetten). Alternativ koennten wir die nachtraegliche Aenderung der Buffergroesse natuerlich verbieten, was aber ein wenig die Flexibilitaet raubt.

2. Wir lassen eine variable Anzahl von Elementen pro Block zu. Hierfuer muessten wir natuerlich fuer jeden Block Zusatzinformationen ueber die Anzahl der enthaltenen Elemente speichern. Es waere uns hiermit zwar von vornherein unmoeglich das absolute Optimum zu erreichen, aber da diese Information sich auf 2Byte begrenzen lassen wuerde, waere dies wohl noch zu verkraften. Das viel groessere Problem ergibt sich aus der schwereren Verwaltung und Unvorhersehbarkeit der auszulesenden Elemente. Im WorstCase koennte es passieren, dass wir bei jedem Einlagerungsvorgang nur ein Element lesen oder mehr Elemente als wir in den Buffer uebertragen koennen und somit die verbleibenden Elemente wieder zurueck auf den externen Speicher schreiben muessten.

Egal wie man sich letztenendlich entscheidet, keine der beiden Varianten erweisst sich als optimal, noch als leicht zu Programmieren. Bei dem Blockbasiertem internen Buffer werden wir dieses Problem recht geschickt umgehen.
Zwar bieten uns itembasierte Buffer flexibel verwaltbaren Speicher auf dem sich einige Optimierungen durchfuehren lassen, die im nachfolgend erklaertem blockbasierten Buffer nicht (oder nur schwer) moeglich sind[1], aber unterliegt diesem in den besonders wichtigen Optimierungen auf unserem speziellem Interface. Allgemein ist der itembasierte Buffer zu waehlen, wenn man eine mittlere Anzahl von Datenstruktur braucht und sich die konstanten 2 Bloecke RAM pro Stueck nicht leisten kann. Es ist jedoch zu beachten, dass wie beim externen Buffer temporaerer Speicher gebraucht wird und diese Variante sehr rechenaufwendig sein kann[2].

3.2.2 Blockbasierte interne Buffer

Der blockbasierte interne Buffer ist verwandt mit den Pagingsystemen von Betriebssystemen. Anstatt wie beim itembasierten Buffer einzelne Elemente zu buffern, haben wir eine Anzahl von Bloecken im Hauptspeicher, die Bitkompatibel zu denen im externen Speicher sind. Auf diese Weise koennen wir auf direktem Weg (ohne umzukopieren) Elemente mit dem externen Speicher austauschen. Das Erstellen eines temporaeren Blocks im Hauptspeicher entfaellt also. Da wir die RAM-Bloecke in einer Reihe im Hauptspeicher erstellen koennen, koennen wir auch ohne Speicherprobleme mehrere Bloecke auf einmal austauschen und somit beim Auslagern unter Umstaenden sehr viel Zeit einsparen. Desweiteren koennen wir auf diese Weise die Bloecke vollstaendig ausnutzen und

[1] zum Beispiel eine dynamische Unterteilung des Buffers in Lese- und Schreibebuffer

[2] Dies ist nicht zu unterschaetzen. Da die Datenstrukturen mit sehr grossen Datenmengen hantieren koennen sollen, brauchen wir oft 32 oder 64Bit Variablen, die auf Mikrokontrollern oft so langsam sind, dass sie die IO-Zeiten zum kleinsten Problem machen.

somit die Elemente mit minimalen IOs verschieben. Allerdings kann auf diese Weise Verschnitt im Buffer entstehen, falls die Groesse der Elemente nicht ganz in die Blockgroesse aufgeht und Elemente die groesser als ein Block sind lassen sich erst garnicht speichern (Elemente die auf zwei Bloecke aufgespalten werden, lassen sich unter Umstaenden nur mit hohem Aufwand wieder auslesen). Allgemein arbeitet man jedoch mit Elementen der Groesse eine 2er-Potenz die kleiner als ein Block von typischerweise 512Byte sind und somit glatt aufgehen. Mit blockbasierten internen Buffern sind wir in der Lage die maximale IO-Leistung zu erreichen. Leider braucht die Queue aber mindestens 2 Bloecke Buffer [3] und der Stack einen, wobei mit nur einem Block Buffer Seitenflattern auftretten kann [4]. Im Praxistest mit einem Arduino Mega hat sich gezeigt, dass es fuer jeden weitern Block im Buffer nur wenige Prozent Leistungssteigerung gibt[5].

4 Optimierungen

4.1 Basic - Work on Buffer

Der Vollstaendigkeit halber wollen wir die grundlegende Optimierung hinter dem Buffersystem erlaeutern:
Da unser Buffer bedeutend schneller ist als der externe Speicher und keine IOs verbraucht, sollten wir natuerlich moeglichst so viel auf dem Buffer operieren wie moeglich. Gerade beim Stack ist dies sehr einfach, da wir immer aus der gleichen Richtung Lesen und Schreiben. Wir brauchen also nur die letzten Elemente im Buffer halten und nur Aus oder Einlagern wenn der Buffer voll oder Leer ist. Auf diese Weise werden die meisten Elemente vermutlich nie auf den externen Speicher ausgelagert werden muessen und benoetigen damit keinen einzigen IO. Fuer die Queue ist es etwas komplizierter, da wir an das Ende schreiben und vom Anfang lesen. Wir brauchen also einen Schreibe- und einen Lesebuffer, solange alle Elemente aber noch in den Buffer passen, koennen diese Vereinigt werden und brauchen auch hier keine IOs. Sobald die Anzahl der Elemente aber die Groesse des Buffers uebersteigt muessen wir eine Entscheidung treffen, wie der Buffer aufgeteilt wird. Dieses Problem exisitiert aber nur beim Itembasierten Buffer, da beim Blockbasierten Buffer ein Lesebuffer nur einen Block braucht [6].

[3] Einen zum Schreiben(Ende) und einem zum Lesen(Anfang). Die Bloecke in der Mitte werden ausgelagert

[4] Ab zwei Bloecken ist dies nichtmehr der Fall

[5] Bei schnelleren Systemen kann dies jedoch anders aussehen. Aufgrund des beschraenkten Hauptspeichers, konnte nur bis zu einer niedrigen Buffergroesse getestet werden.

[6] Siehe partielles Auslagern

IOs	Buffer						Operationen
							PUSH(1,2,3,4,5)
	1	**2**	**3**	**4**	**5**		POP(): 5
	1	**2**	**3**	**4**			PUSH(6,7)
	1	**2**	**3**	**4**	**6**	**7**	PUSH(8)
Buffer auf SD auslagern							
	8						POP(): 8
							POP()
Einen Block einlagern	**6**	**7**					: 7
	6						

Beispielablauf eines blockbasierten Stacks. 11 Operationen bei 4 IOs.

4.2 Partielles Auslagern

Da wir bei internen Buffern grundsaetzlich alle Operationen ueber den Buffer laufen lassen, stellt sich die Frage des optimalen Ein- und Auslagern mit dem externen Speicher. Wenn wir Lesen wollen, so ist ein voller Buffer wuenschenswert. Wenn wir hingegen Schreiben wollen, sollte dieser moeglichst Leer sein damit wir das Ein- bzw Auslagern soweit wie moeglich hinauszoegern koennen. Fuer den Fall, dass wir nicht wissen was als naechstes getan werden soll, waere irgendwas dazwischen zu empfehlen. Gerade beim itembasierten Buffer haben wir hier sehr viel Optimierungspotential. Beim blockbasierten hingegen haben wir gewoehnlich nur sehr wenige Blocke Buffer und somit nur wenige Moeglichkeiten.
Das Problem ist nun, wie wir herrausfinden, wie weit wir unseren Buffer fuellen oder leeren muessen um so effizient wie moeglich zu sein. Eine Moeglichkeit waere einen Standardwert zu haben, den der Anwender je nach Anwendung evtl. selber modifizieren kann um so eine Effizienzsteigerung zu erzielen. Alternativ koennten wir auch die letzten Operationen betrachten um so das vorraussichtlich passende Verhaeltnis zu schaetzen. Hierbei sollte man aber ein speichersparsammes Verfahren verwenden. Wenn man jede Operation loggen wuerde, so braeuchten wir pro betrachteter Opertation mindestens ein Bit. Fuer eine brauchbare Schaetzung kann somit eine Menge Speicher verschwendet werden, die aufwendige Berechnung nicht mitgezaehlt. Eine einfachere Variante waere es, die Anzahl der Operationen seit z.B. dem letzten Lesezugriff zu zaehlen.
Allgemein machen die speziellen Eigenschaften unseres Blockinterfaces diese Optimierungen aber sehr schwer. Da der Aufwand des Auslagerns abhaengig von der Anzahl der auszulagernden Bloecken, aber immer deutlich aufwendiger als Einlagern ist, kann man nur schwer ein gutes Verhaeltnis bestimmen. Es empfiehlt sich daher den zu leerenden Buffer immer komplett auszulagern, da der Vorteil des schnelleren Auslagern groesser ist als der Nachteil des evtl. anschliessenden sofortigen Einlagerns. Beim Einlagern sollte immer nur ein Block eingelagert werden, da der Zeitaufwand unabhaengig der Anzahl der gleichzeitig einzulagernden Bloecke ist. Auf diese Weise halten wir aber mehr Platz fuer neue Elemente im Buffer.

4.3 Buffer recycling

Desweiteren kann man Ausnutzen, dass wir zumindestens beim Stack die evtl. wieder einzulagernden Seite nicht sofort ueberschreiben und somit in dem Fall

den sonst wieder einzulagernden Block wiederherzustellen. Genauso brauch man die wiederhergestellten Bloecke nicht erneut auszulagern. Diese Optimierung laesst sich beim Stack mit blockbasiertem internen Buffer mithilfe von 2 8Bit Integern verwirklichen und ist somit sowohl Speicher als auch Rechensparsem. Im Praxistest konnten auf diese Weise wenige Prozent an Einlagerungen gespaart werden, bei gleichbleibenden Auslagerungen und Rechenzeit.

Bemerkung	Buffer									Operation
Buffer voll	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	PUSH(10)
Buffer leeren	**10**			*4*	*5*	*6*	*7*	*8*	*9*	POP(): 10
Buffer ist leer				*4*	*5*	*6*	*7*	*8*	*9*	POP()
Wiederherstellen	**4**	**5**	**6**	**7**	**8**	**9**				: 9
	4	**5**	**6**	**7**	**8**					PUSH(11,12,13,14)
	4	**5**	**6**	**7**	**8**	**11**	**12**	**13**	**14**	PUSH(15)
Bereits auf SD	*4*	*5*	*6*	**7**	**8**	**11**	**12**	**13**	**14**	
Verschieben	**7**	**8**	**11**	**12**	**13**	**14**				
	7	**8**	**11**	**12**	**13**	**14**	**15**			

Sparen von IOs durch Bufferrecycling bei einem blockbasierten Stack

4.4 Buffer sharing

Das Unterteilen des Queue Buffers in einen festen Lese und Schreibebuffer, bringt natuerlich den Nachteil, dass der Schreibebuffer unter Umstaenden ausgelastet ist, waehrend der Lesebuffer leer ist. Theoretisch koennten wir nun kurzfristig den Speicher des Lesebuffers dem Schreibebuffer zuteilen und so das Auslagern ein wenig hinauszoegern bzw. einen Block mehr Auslagern schreiben (siehe Eigenschaften des Blockinterfaces). Wir erhalten eine Optimierung, wenn der Lesebuffer leer ist und waehrend wir den Lesebuffer geliehen haben, keine Leseoperationen eingehen. Andernfalls muessten wir den Schreibebuffer leeren und den partiellen Block im Lesebuffer in den Schreibebuffer kopieren. Auf schnellen Geraeten, bei denen der externe Speicher der einzige Flaschenhals ist, wuerde sich diese vergleichsweise kleine Optimierung wohl lohnen, da diese Optimierung nur geringe Verwaltung und gelegentlich das Verschieben von Speicher braucht. Auf den langsamen Arduinos waere diese Optimierung allerdings zu klein um die Kosten zu decken.

5 Persistenz

Um Persistenz in konstantem Code zu gewaehrleisten muessen alle Instanzvariablen bei der Initalisierung aus dem externen Speicher geladen und bei der Zerstoerung wieder dahin geschrieben werden. Um das Laden aus Datenmuell zu verhindern, ist ein ValidierungsCode zu empfehlen, der testet ob die geladenen Variablen nicht eventuel inkonsistent sind und die neugeladene Datenstruktur damit undefiniert waere. Des weiteren bietet es sich an, auch alle konstanten Werte, die nicht geaendert werden duerfen (wie den ersten und den letzten reservierten Block auf dem externen Speicher), sollten abgespeichert und beim Laden ueberprueft werden.
Desweiteren sind partielle Bloecke, bei denen es nicht zu einem vollstaendigen

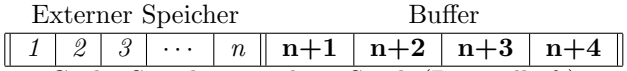
Block gereicht hat, bei Konstruktion in den Buffer zu laden bzw. bei Zerstoerung zu sichern.
Die Persistenzeigenschaft soll ein Vorteil nich ein Zwang sein. Daher sollten wir neben einer Loeschoperation auch des Deaktivieren der Persistenz ermoeglichen. Ein Parameter im Konstruktor, der das Neuerstellen erzwingt, ist eine sehr Nutzerfreundliche und effiziente Umsetzung.
Durch die Persistenzeigenschaft muessen wir die Datenstrukturen nicht die ganze Zeit im Hauptspeicher halten. Es lassen sich unzaehlige Datenstrukturen parallel verwalten, indem man diese nur erzeugt wenn sie gebraucht werden und anschliessend sofort wieder zerstoert. Es lohnt sich hierbei natuerlich nicht die Datenstruktur nur fuer eine Operation wiederherzustellen, da ein Wiederherstellungs und Zerstoerungsvorgang bis zu 4 IOs brauchen kann.

6 Implementierung

Kommen wir nun endlich zu den im Algorithmikpraktikum entstandenen Implementierungen. Wie bereits angesprochen, habe ich auf Blockbasierte interne Buffer gesetzt um somit eine sehr IO-arme und schnelle Umsetzung bieten zu koennen. Dies geht zwar auf Kosten des Hauptspeichers, aufgrund des relativ hohen Minimalbedarf von 2 Bloecken, aber durch geschickte Ausnutzung der Persistenz kann man auch mit knappen Ressourcen gut haushalten.
Sowohl der Stack als auch die Queue erreichen das optimale IO-Verhalten, welches bei $1/(\text{BLOCKSIZE}/sizeof(T))$ Schreibeoperationen pro Einfuegen bzw. Leseoperationen pro Auslesen liegt. In der Praxis geht nur etwa 30% der Rechenzeit auf die IOs, der Rest auf Operationen mit Long-Integers.
Um das Dokument nicht ausarten zu lassen, werden die beiden Datenstrukturen auf die wichtigsten Punkte reduziert. Fuer die genaue Implementierung sei auf den kommentierten Sourcecode verwiesen.

6.1 Stack

Im Stack halten wir immer die letzten BUFFERSIZE-Bloecke im Hauptspeicher, die restlichen auf dem externen Speicher. Wir koenne also immer direkt aus dem Buffer lesen/schreiben. Sollte der Buffer leer sein, so Lagern wir einen Block ein. Sollte der Buffer voll sein, so leeren wir ihn komplett. Durch Bufferrecycling mithilfe von 2 8Bit Integern werden einige IOs gespart.
Der Buffer selbst ist ein Stack mit festem Anfang, der nach oben waechst. Bei Bufferrecycling muessen wir die Bloecke an den Anfang schieben. Nur so ist es moeglich, beim Auslagern alle Bloecke an einem Stueck zu schreiben. Die Bloecke auf dem externen Speicher entsprechen wieder einem Stack mit festem Anfang. Der Stack ist voll, sobald im Stack auf dem externen Speicher der hoechste zugeteilte Block erreicht wurde.
Eine grobe Funktionsweise ist in den Darstellungen in der Sektion 'Optimierungen' zu sehen.



Grobe Speichereinteilung Stack (Beispielhaft)

6.2 Queue

Die Queue ist etwas komplizierter als der Stack, da wir an beiden Enden einen Buffer brauchen. Der Lesebuffer besteht aus genau einem Block der selber als Queue mit rotierendem Anfang verwaltet wird. Da der Lesebuffer nicht gefuellt wird, muss er nur bei Zerstoerung der Queue ausgelagert werden. Der Schreibebuffer entspricht einem Stack mit festem Anfang um an einem Stueck auf den externen Speicher geschrieben werden zu koennen. Die Bloecke auf dem externen Speicher sind als Queue mit rotierendem Anfang organisiert. Die Queue ist voll, wenn die Anzahl der beschriebenen Bloecke der Differenz zwischen niedrigsten und hoechstem zugeteilten Block entspricht.
Solange keine Bloecke auf den externen Speicher ausgelagert sind, wird der Lesebuffer moeglichst intensiv benutzt. So wird geprueft, ob ein einzufuegendes Element gleich in den Lesebuffer kann oder ob man anstatt den vollen Schreibebuffer auszulagern, einige Elemente in den Lesebuffer schieben kann um ein wenig Platz zu schaffen.
Es ist durch die rotierende Umsetzung im externen Speicher, der Moment zu beruecksichtigen, indem wir am Ende des zugeteilten Speicherbereichs angelangt sind und am Anfang weitermachen muessen. Hierbei werden nur die noch an das Ende passenden Bloecke geschrieben und die restlichen im Buffer nach vorne geschoben. Sollte der Buffer wieder voll werden, so wird am Anfang des zugewiesenen Speichers fortgefahren.

Lesebuffer			Externer Speicher					Schreibebuffer				
1	**2**	**3**	*4*	*5*	$\cdots$	*n*	*n+1*	**n+2**	**n+3**	**n+4**	**n+5**	**n+6**

Grobe Speichereinteilung Queue (Beispielhaft)

Beispielablauf: (Lesebuffer[1 Block], Schreibebuffer[1 Block], Externer Speicher)

	L		S		E						Operation
Start	**1**	**2**	**3**	**4**							POP(): 1
		2	**3**	**4**							POP(): 2
			3	**4**							POP()
Verschieben	**3**	**4**									: 3
Auslesen		**4**									PUSH(5,6,7)
	5	**4**	**6**	**7**							PUSH(8,9)
Auslagern	**5**	**4**			*6*	*7*					
Einfuegen	**5**	**4**	**8**	**9**	*6*	*7*					PUSH(10)
Auslagern	**5**	**4**			*6*	*7*	*8*	*9*			
Einfuegen	**5**	**4**	**10**		*6*	*7*	*8*	*9*			POP(): 4
	5		**10**		*6*	*7*	*8*	*9*			POP(): 5
			10		*6*	*7*	*8*	*9*			POP()
Einlagern	**6**	**7**	**10**				*8*	*9*			: 6
Auslesen		**7**	**10**				*8*	*9*			POP(): 7
			10				*8*	*9*			POP()
Einlagern	**8**	**9**	**10**								: 8
		9	**10**								POP(): 9
			10								POP()
Verschieben	**10**										: 10
Leer											