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 daher 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 moeglichst schnelle und persistente Datenstrukturen zu haben.

2 Eigenschaften des Blockinterfaces

Das von der Wiselib verwendete Blockinterface 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 512Byte. Es koennen auch mehrere zusammenhaengende Bloecke sowohl gelesen als auch geschrieben werden. Der hierbei verwendete RAM-Block muss dabei natuerlich zusammenhaengend sein.
- Lesen geht etwa doppelt so schnell wie schreiben. Das Lesen eines Blockes dauert etwa 2ms. Diese Zeit verkuerzt sich auch durch das Lesen mehrere Bloecke am Stueck nicht.
- Das Schreiben eines Blockes braucht etwa 4ms und hat die unangenehme Eigenschaft, dass jeder 128 Schreibvorgang um die 200ms braucht. Dafuer ist die durchschnittliche Schreibedauer fuer mehrere Bloecke um bis zu 50% kuerzer.

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 arbeitet
2. Einen Itembasierten Internen Buffer, der eine bestimmte Anzahl von Elementen buffert.
3. Einem Blockbasierten Internen Buffer, der ganze Bloecke im RAM haelt und mit dem ExternalMemory 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.

3.1 Externe Buffer

Die Datenstruktur verfuegt nicht ueber einen eigenen Buffer sondern ihm werden gefuellte Bufferkontainer uebergeben und es wird dann nur versucht, diese mit moeglichst wenigen IOs auf die SD zu bringen oder auszulesen. Als einfachste Beispiel kann ein Array uebergeben werden und der Algorithmus versucht so viele Elemente wie moeglich in einen Block zu schreiben. Auf diese Weise kann sich das IO-Verhalten im Vergleich zum einzelnen Einfuegen und Auslesen von Elementen deutlich verbesseren, aber es stellt fuer den Verwender der Datenstruktur einen erheblich hoeheren Programmieraufwand da und ist nur bei richtiger Andwendung effizient. Obwohl wir auf diese Weise eine recht flexibele Buffer/Speicher-Verwaltung haben, ist sie den anderen Varianten durch ihre komplizierte Anwendung unterlegen.
In dieser Variante fuehrt jede Operation zu mindestens einem IO, vermutlich jedoch eher zu zweien, da wechselnde Buffergroessen partielles Schreiben von Bloecken (damit ist das Fuellen von Bloecken mit einer variablen Anzahl von Elementen gemeint) noetig machen kann. Dies liesse sich verhindern, indem man variable Fuellmenge bei Bloecken zulaesst, indem man ihnen die Information ueber die Anzahl der enthaltenen Elemente gibt. Beim anschliessenden Wiederauslesen kann dies aber zu einer uebermaessigen Anzahl von Lesevorgaengen fuehren, da jeder Lesevorgang eventuell nur eine minimale Anzahl von Elementen liefert.
Insofern es nicht weitere Hindernisse fuer den anwendenden Entwickler geben soll, muss fuer jeden IO ein Block temporaer im RAM angelegt werden.

3.2 Interne Buffer

3.2.1 Itembasierte interne Buffer

In dieser Buffervariante wird ein statischer Buffer in Form eines Arrays ueber N Elementen in der Datenstruktur angelegt. Saemmtliche Operationen laufen hierbei ueber den Buffer, welcher nach Bedarf gefuellt oder geleert wird. Sollte man ein Element einfuegen wollen, so wird in der Einfuegeoperation getestet, ob noch genug Schreibeplatz[1] vorhanden ist. Sollte dies nicht der Fall sein, so wird eine Funktion aufgerufen die neuen Schreibeplatz durch Umlagern auf den Extern Speicher oder den Leseplatz schafft. Der Lesevorgang verlaeuft relativ analog: Sollte der Leseplatz leer sein, so wird eine Funktion aufgerufen, die Elemente von der extern Speicher oder dem Schreibeplatz umgelagert.
Durch den Buffer ist es uns moeglich Elemente fuer IOs effizient zu sammeln. Es ist jedoch relativ schwer bis teils unmoeglich direkt vom Buffer auf den Externen Speicher zu schreiben oder umgekehrt von diesem in den Buffer zu lesen. Wir muessen einen temporaeren Block anlegen in den wir die Daten von der SD kopieren und anschliessend die einzelnen Elemente daraus in den Buffer schieben (fuer den Lesevorgang analog). Dies wird dadurch bedingt, dass wir nicht davon ausgehen koennen, dass Blockdaten und Bufferdaten Bit Kompatibel sind (Das einfachste Beispiel hierfuer ist, der Fall, in dem der Buffer kleiner ist als ein Block. Der Forderung nach Buffern die groesser sind als die Blockgroesse wird sich im Blockbasierten internen Buffer deutlich einfacher und

[1] Bei der Queue unterscheiden sich Schreibe und Leseplatz, sodass auch wenn der Buffer nicht voll ist, kein Schreibplatz vorhanden sein muss. Teils laesst sich hier aber effizient umordnen.

effizienter angenommen.). Der Geschwindigkeits-Vorteil durch das Schreiben mehrerer Bloecke wird aufgrund des dadurch entstehenden sehr hohen RAM-Verbrauchs getruebt. Da wir nicht Testen koennen, wie viel RAM noch vorhanden ist, koennte dies bei Microcontrollern zu einem Russisch Roulette Spiel werden.
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 [2]. 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 mehr oder weniger Speicherplatz zur Verfuegung steht. Sollte der Buffer naemlich kleiner werden, so koennte es passieren, dass die Bloecke nichtmehr vollstaendig in den Buffer geladen werden koennten und wir partielles Lesen und Schreiben benutzen muessten. 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. Dies wuerde uns natuerlich auch dazu zwingen in jeden Block auch die Anzahl der enthaltenen Elemente zu schreiben(was aber lediglich ein wenig mehr Programmieraufwand und eine um 2 Byte verringerte Blockgroesse verursachen wuerde). Das groessere Problem ist aber die kompliziertere Verwaltung und es waeren nur bestimmte Buffergroessenveraenderungen moeglich (Sollte der Buffer kleiner sein, als die im Block enthaltenen Daten, so muesste er den Block auf dem Extern Speicher modifizieren, da er ja schon Teile gelesen ausgelesen hat.). Einen weiteren Nachteil werden wir gleich beim Buffer-ExternalMemory-Exchange sehen.

Keine der beiden Optionen erweist sich weder als optimal noch als einfach zu Programmieren. Bei dem Blockbasiertem internen Buffer werden wir dieses Problem recht geschickt umgehen.
Im Allgemeinen ist der Itembasierte interne Buffer die beste Wahl, wenn wir dem Blockbasiertem internen Buffer keine 2 Bloecke RAM statisch zur Verfuegung stellen koennen. Auch wenn er bei optimaler Programmierung und Vergleichbarer Buffergroesse annaehrend so effizient sein kann wie der Blockbasierte Interne Buffer so ist diese Variente aufgrund seiner komplexen Struktur nur bei geringen RAM-Ressourcen und/oder mehreren parallelen Datenstrukturen zu empfehlen, bei denen man nicht statischen 1kB pro Datenstruktur entbehren kann.

[2] Dies waere zwar kein Weltuntergang, aber wir wollen das Optimum.

3.2.2 Blockbasierte interne Buffer

Der blockbasierte interne Buffer ist stark mit dem itembasierten internen Buffer verwandt, anstatt eines Arrays mit N Elementen wird jedoch ein Array von M zusammenhaengenden Bloecken angelegt (data_ unit_ t[$M * 512$]). Beim Einfuegen von Elementen wird nun ein Block nach dem anderen so gefuellt, wie er auch auf der SD spaeter aussehen soll. Sollten alle Bloecke voll sein, so werden einfach ein paar auf die SD ausgelagert und so neue leere Bloecke geschaffen. Aehnlich wird beim Auslesen auch einfach nur aus den Bloecken gelesen und sollten alle RAM-Bloecke leer sein, so werden einfach ein paar neue (volle) von der SD geholt. Wir lagern also nur noch Bloecke und nicht einzelne Elemente an und aus. Auf diese Weise koennen wir die Kapazitaet der Bloecke natuerlich vollstaendig[3] ausnutzen und der SD-Buffer-Exchange ist sehr einfach und effektiv. Es ist das schreiben von mehreren Bloecken am Stueck moeglich und es muessen fuer die IOs keine temporaeren Bloecke im RAM angelegt werden, da die IOs direkt ueber den Buffer gehen koennen.
Mit blockbasierten internen Buffern sind wir in der Lage die maximale IO-Leistung zu erreichen. Leider brauchen sie aber mindestens 2 Bloecke um effektiv zu arbeiten. (Jeder weitere Block fuehrt zu mehr Bloecken je Schreibeoperation. Der Stack kann zwar theoretisch bereits ab einem Block arbeiten, aber hierbei kann es zu einem im ExternalMemory-Buffer-Exchange besprochenen WortCase kommen.)

3.2.3 ExternalMemory-Buffer-Exchange

Da wir bei den internen Buffern grundsaetzlich alle Operationen direkt ueber den Buffer laufen, stellt sich hier das Problem des effizienten Austausches der Daten zwischen Buffer und ExternalMemory. Wenn wir viel Lesen wollen, so sollte der Lesebuffer moeglichst voll sein, wenn wir hingegen viel Schreiben wollen sollte der Schreibebuffer moeglichst leer sein, wenn wir aber wiederrum nicht wissen was wir als naechstes Tun werden sollte es irgendwas dazwischen sein (Im theoretischen Fall, dass Lesen und Schreiben auf dem ExternalMemory gleich lange dauern, so sollte der Buffer zu 50% gefuellt sein. Wir werden den Parameter der gewuenschten Bufferfuellmenge im Folgendem als P bezeichnen.). Dieses Problem ist bei dem Itembasierten internen Buffer, wie wir schon gesehen haben, deutlich komplexer, aber es laesst unter Umstaenden auch mehr Optimierungspotential. Beim Blockbasiertem internen Buffer wird uns schnell Auffallen, dass dieses Problem fuer den Lesevorgang wegfaellt. Da das Lesen eines Blockes, unabhaendig von der Anzahl der am Stueck zu lesenden Bloecken, konstante Zeit dauert, ist es vorzuziehen grundsaetzlich nur einen Block zu lesen wenn der Buffer leer ist. Sollten wir nur Lesen wollen, so wuerde sich in der Laufzeit nichts aendern. Wenn wir aber plotzlich doch schreiben muessen, so haben wir noch viel mehr Platz im Buffer. Bei itembasierten internen Buffer mit variabler Elementanzahl pro Block, koenne wir lediglich das Maximum bestimmen.
Bei Itembasierten internen Buffer haben wir allgemein das Problem, dass wir den Parameter zu den die Bloecke gefuellt oder geleert werden sollen, nur sehr

[3] Vollstaendig ist hierbei vielleicht ein wenig Uebereifrig, da dies nur gilt wenn die Groesse des generischen Types in den Block aufgeht. Ein Element auf 2 Bloecke zu splitten koennte den SD-Buffer-Exchange erschweren.

schwer aendern koennen, da wir im Buffer mit Elementen arbeiten und auf dem ExternalMemory mit Bloecken, bei denen wir bei variabler Fuellmenge noch nichteinmal Wissen wie viele Elemente in einem Block sind. Sollten die gewuenschte Anzahl von Elementen im Buffer nicht zur Blockfuellmenge passen, koennte der Lesevorgang mehrere IOs brauchen.
Kommen wir zurueck zur Optimalen Fuellmenge. Wir werden jetzt als Beispiel den WorstCase fuer den Blockbasierten internen Buffer fuer den Stack mit nur 1 Block Buffer ansehen:
$\cdots ToDo \cdots$
Sehen wir uns allgemein die beiden ExtremVarianten an. Wir gehen hierbei erstmal davon aus, dass jeder IO die gleiche Laufzeit hat.

1. Wir fuellen und leeren den Buffer immer genau zur Haelfte. Wenn der Nutzer anschliessend einige Elemente Lesen will, so muss erst 0.5*BUFFERSIZE Leseoperationen ein ExernalMemory-Buffer-Exchange ausgefuehrt werden. Analog gilt dies fuers Schreiben.

2. Wir fuellen und leeren den Buffer immer vollstaendig. Wenn der Leser nun sehr viel Lesen will, so kann er bis zu BUFFERSIZE Elemente lesen bevor es zum naechsten ExternalMemory-Buffer-Exchange kommt. Analog verlaeuft es wieder beim Schreiben. Leider kriegen wir aber auch den WorstCase den wir vorhin uns angesehen haben.

Da wir nur schwer vorhersagen koennen, ob der Nutzer als naechstes sehr viel Lesen, Schreiben oder beides in gleichen Masse tun wird, ist es sinnvoll einen Standardparameter festzulegen, wobei wir wenn moeglichst beruecksichtigen sollten, dass ein Schreibender ExternalMemory-Buffer-Exchange teurer ist als ein Lesender.
$\cdots$ ToDo Berechnung des optimalen Standardparameter unter beruecksichtung des teureren Schreibevorgangs $\cdots$
Es waere jedoch auch aeussert praktisch fuer den Nutzer, wenn er den Parameter selbst bestimmen koennte (Wir wollen ihm das natuerlich aber nicht aufzwingen. Um die berechnung eines Defaultparameters kommen wir also nicht herum.). Unter Umstaenden waere es aber auch verdammt Cool, einfach die letzten vom Nutzer ausgefuehrten Operationen zu loggen und den Parameter auf diesen Daten flexibel zu bestimmen. Auf diese Weise waere koennten wir durchschnittlich einen sehr guten und flexibelen ExternalMemory-Buffer-Exchange erreichen[4].

4 Die Datenstrukturen

Nachdem wir uns die Optimierungen durch Buffer angesehen haben, kommen wir nun zu den eigentlichen Datenstrukturen, dem Stack und der Queue. Als Bufferkonzept habe ich mich, wie bereits angekuendigt, fuer den Blockbasierten internen Buffer entschieden. Neben der Tatsache, dass er leichter zu Implementieren ist als ein Itembasierter, koennen die Eigenschaften des Blockinterfaces voll ausgenutzt werden. Der Speicherverbrauch von mindestens 1kB ist zwar

[4]Das vorgestellte WorstCase Szenario koennte hierbei zwar kurzfristig auftauchen, aber durch geschickte Programmierung sollte dies feststellbare sein und es koennen automatisch die Parameter angepasst werden.

fuer Microcontroller recht hoch, fuer wenige sehr kleine gar zu hoch, sollte aber fuer die meisten den Preis wert sein, insofern sie nicht mehrere parallel benutzen wollen[5] [6].

4.1 Stack

Unter Beachtung der in der Sektion 'Buffer' ist ein Stack nicht sonderlich schwer zu implementieren. Die Bloecke auf der SD lassen sich als Stack verwalten, der nur in eine Richtung waechst und nicht wandert[7]. Im Speicher selbst haben wir eine Queue, da wir den aelteren Teil des Stacks auslagern aber von der anderen Seite auffuellen. Da eine herkoemmliche Queue wandert und wir somit einen Speicherbruch[8] bekommen koennen, der ein Schreiben am Stueck verhindert, empfehlt es sich zu ueberlegen ob ein verschieben der Buffer nach Auslageroperationen aufgrund des schnellen Speichers zu bevorzugen ist.

4.2 Queue

Fuer die Queue brauchen wir ansich zwei Buffer. Einen zum Lesen am Anfang der Queue und einen zum Schreiben am Ende der Queue. Die Daten in der Mitte koennen ausgelagert werden. Waehrend sich die Entscheidung der Aufteilung des zur Verfuegung gestellten Buffers beim Itembasierten internen Buffer als aeusserst kompliziert erweisen koennte[9], ist es beim Blockbasierten intenen Buffer sehr einfach: Wir nehmen grundsaetzlich nur einen Block als Lesebuffer, da wir bereits im fruehrem Teil bemerkt haben, dass das Buffern von mehreren Lesebloecken nachteilig ist. Auf der dem ExternalMemory ist noch zu berueckksichtigen, dass die benutzten Bloecke wandern und brechen koennen (Wenn wir zu einer Speichergrenze im RAM kommen muessen wir am Anfang des Speichers fortfahren.). Fuer weitere Informationen sei auf den Implementierungspart in Pseudocode verwiesen.

5 Persistenz

Um Persistenz in konstantem Code zu gewaehrleisten muessen alle Instanzvariablen bei der Initalisierung aus dem ExternalMemory geladen werden 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 eventuell inkonsistent sind und die neugeladene

[5] Aufgrund des Persistenz liesse sich das aber auch machen. Wir zerstoeren einfach die alte Version und erstellen eine neue. Sobald wir mit der fertig sind, laden wir die alte neu. Da fuer das Erstellen eines Datentyps 3(Queue) bzw 2(Stack) Leseoperationen und fuer das Zerstoeren wieder 3(Queue) bzw. 2(Stack) Schreibeoperationen noetig sind, sollte man dies aber nur gezielt anwenden

[6] Damit wird sich auf den konstanten Speicherverbrauch bezogen. Mit den Itembasierten internen Buffern liesse sich ein Speicher von wenigen Bloecken auf sehr viele Datenstrukturen verteilen. Ihren temporaeren Block wuerden sie nie gleichzeitig brauchen.

[7] Das Problem des 'Wanderns' stellt sich spaeter in der Queue

[8] 4 Bloecke sollen ausgelagert werden, die ersten 3 werden noch ans Ende geschrieben, der 4. an den Anfang.

[9] Eine sehr einfache Variante waere jedoch auch einfach eine konstante 1 zu 1 Aufteilung, die zwar recht gut sein sollte, aber nicht perfekt.

Datenstruktur damit undefiniert waere. Des weiteren bietet es sich an, auch alle konstanten Werte, die nicht geaendert werden duerfen (BUFFERSIZE darf Beispielsweise beim Blockbasierten internen Buffer geaendert werden), 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 Alternative zur Loeschoperation.

6 Implementierung in Pseudocode

Wir beschraenken uns auf die zwei Wichtigsten Operationen Push und Pop. Der Pseudocode unterscheidet sich syntaktisch stark von der tatsaechlichen Implementierung[10], aber ist Semantisch relativ aequivalent.

6.1 Stack

Der Buffer des Stacks ist sehr einfach aufgebaut: Es wird Elemente ausschliesslich am Ende angefuegt oder entfernt. Der Programmieraufwand um die Daten Konsistent zu halten ist verhaeltnismaessig gering.

6.1.1 Push

```
BOOL PUSH(element)
    IF(Freie Bloecke im ExternalMemory < BUFFERSIZE) RETURN FALSE
    IF(buffer voll?)
        berechne nach externalMemory zu schreibende Bloecke -> N
        schreibe die ersten N Bloecke des Buffers zum externelMemory
        verschiebe die uebriggebliebenen Bloecke des Buffers zum Bufferanfang
    schreibe element an bufferende
    RETURN TRUE
```

6.1.2 Pop

```
BOOL PUSH(*element)
    IF(buffer leer)
        IF(externalMemory leer) RETURN FALSE
        hole den letzten Block von externelMemory und schreibe ihn an den Anfang des Buffers
    *element = letztes Element aus Buffer
    RETURN TRUE
```

[10] Damit ist nicht nur die triviale Beobachtung gemeint, dass PseudoCode $\neq$ C++, sondern eher das die Struktur voellig anders ist.

6.2 Queue

Der Buffer besteht aus mindestens zwei Bloecken. Der erste Block ist der Read-Buffer, die anderen der schreibebuffer. Der Readbuffer ist als eine Queue aufgebaut, waehrend die Element im Schreibebuffer bei Bedarf verschoben werden.

6.2.1 Push

```
BOOL PUSH(element)
    IF(freie Bloecke auf ExternalMemory < BUFFERSIZE) RETURN FALSE
    IF(externalMemory leer AND schreibebuffer leer AND lesebuffer nicht voll)
        schreibe element an das ende des Lesebuffers
    ELSE
        IF(schreibebuffer voll)
            berechne Anzahl der optimal auf den externalMemory zu schreibenden Bloecke -> N
            berechne Anzahl der ohne Speicherbruch auf externalMemory zu schreibenen Bloecke -> M
            MIN(N,M) -> O
            schreibe die ersten O Elemente des SchreibeBuffers auf den ExternalMemory
            verschiebe, falls vorhanden, die hinteren Bloecke des Schreibbuffers zum anfang
        schreibe element an das ende des schreibebuffers
    RETURN TRUE
```

6.2.2 Pop

```
BOOL POP(*element)
    IF(lesebuffer leer)
        IF(externalMemory leer) RETURN FALSE
        hole den ersten Block vom externalMemory und schreibe ihn in den ersten Block des Buffers
    hole erstes Element aus lesebuffer
    RETURN TRUE
```