Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Seminar „Effiziente Programmierung“
Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Speicherverwaltung im Kernel

– Ausarbeitung –

Vorgelegt von:	Sven Schmidt
E-Mail-Adresse:	4sschmid@informatik.uni-hamburg.de
Matrikelnummer:	6647018
Studiengang:	Informatik B.Sc.
Betreuer:	Anna Fuchs; Dr. Michael Kuhn

Abgabedatum: 9. Februar 2017

Inhalt

1 Einleitung

Speicher im Sinne von *Random Access Memory (RAM)*, *Arbeitsspeicher*, ist ein wichtiger Teil eines Computers; er wird daher auch *Hauptspeicher* genannt[13]. Sein primärer Zweck ist es, Teile des *Operating System (OS)* oder von Programmen sowie Daten, die im Moment oder häufig genutzt werden, für den schnellen Zugriff bereitzuhalten. Damit stellt er einen Flaschenhals dar, wenn es um Performance geht: Ohne ihn können Programme nicht arbeiten und mit zu wenig Arbeitsspeicher fühlt sich das System schleppend an. Der RAM bietet wesentlich höhere Zugriffsgeschwindigkeiten, als sekundäre Speichergeräte (Festplatten), bei denen Lese- und Schreiboperationen vergleichsweise teuer[1] sind. Dafür hat Arbeitsspeicher eine weitaus geringere Kapazität.

Während in den meisten Programmiersprachen die Speicherverwaltung keine Rolle für den Programmierer spielt, weil in höheren, stark abstrahierenden Sprachen wie etwa Java oder C# Speicher durch die Verwendung des `new`-Schlüsselworts automatisch reserviert wird, ist dies beispielsweise in C anders. C, das ist eine maschinennahe Sprache[2], deren Sprachkonstrukte sehr effizient auf Maschineninstruktionen abgebildet werden, die dadurch vor allem Zugriff auf den Hauptspeicher in einer sehr tiefen Ebene ermöglichen. Daher eignet C sich besonders für Betriebssysteme: C ist die Programmiersprache des Linux-Kernels, auf den sich diese Arbeit fokussiert. Da Arbeitsspeicher so kostbar und begrenzt ist, ist Effizienz einerseits unter dem Aspekt der „effizienten Speicherverteilung“ (durch den Kernel) zu verstehen, aber in erster Linie soll es darum gehen, wieso speichereffizientes Programmieren für den Linux-Kernel nicht trivial ist und welche Fragen und Probleme damit im Zusammenhang stehen.

2 Überblick

Der Begriff *Kernel* ist kein Synonym für das OS. Vielmehr ist er ein fundamentaler Teil davon; fundamental in dem Sinne, dass der Kernel Dienstleistungen anbietet, auf die der Rest des Betriebssystems aufbaut. Der Kernel ist derjenige Teil des OS', der direkt mit der Hardware interagiert und Kontrolle über das komplette System hat. Er ist die erste Komponente, die während des Systemstarts in den Speicher geladen wird und verbleibt dort bis zum zum Abschalten des Systems. Zu den angebotenen Dienstleistungen gehören vor allem das Prozessmanagement und -scheduling, die Netzwerkverwaltung, das Dateimanagement, *Input/Output (I/O)*, die Hardwareverwaltung (der Kernel beinhaltet die meisten Gerätetreiber), und – hier sehr wichtig – die **Speicherverwaltung**[3].

Wenn wir von Speicher sprechen, so meinen wir den *physischen Speicher*. Das ist der tatsächlich verbaute Arbeitsspeicher auf Hardwareebene. Für den Benutzer ist der Arbeitsspeicher damit vor allem ein Riegel, den er anfassen kann; der Prozessor sieht ihn als fortlaufenden Bereich, in dem jedes Byte eine spezifische *Adresse* besitzt, über die es eindeutig gefunden (*adressiert*) werden kann. Nun wäre es fahrlässig, Prozessen diese Adressen zu verraten, bedenkt man, dass davon nur eine begrenzte Anzahl zur Verfügung steht, um die die Prozesse konkurrieren müssen. Und es ist nichts über die Programme bekannt, die Speicher reservieren wollen: Braucht das Programm wirklich so viel, wie es reserviert? Welche Priorität hat die Speicheranforderung? Hat der Programmierer effizient gearbeitet? Daher braucht es den Kernel als Schicht zwischen der Hardware und den Prozessen, um den Speicher zu verwalten.

[1] *teuer* bezieht sich im Folgenden immer auf verbrauchte Ressourcen, zum Beispiel Zeiteinheiten
[2] vgl. [10, p. 3]
[3] vgl. [11]

Der Kernel organisiert den RAM in so genannten *Pages*. Jede Page hat die – architekturabhängige – Größe `PAGE_SIZE`[4] (in Bytes). Listing 1 zeigt, wie die Größe einer Seite in Erfahrung gebracht werden kann; Abbildung 1 illustriert eine beispielhafte Aufteilung des Speichers in Pages.

```
$ getconf PAGE_SIZE
4096
```

Listing 1: `PAGE_SIZE` kann etwa über `getconf` ausgelesen werden

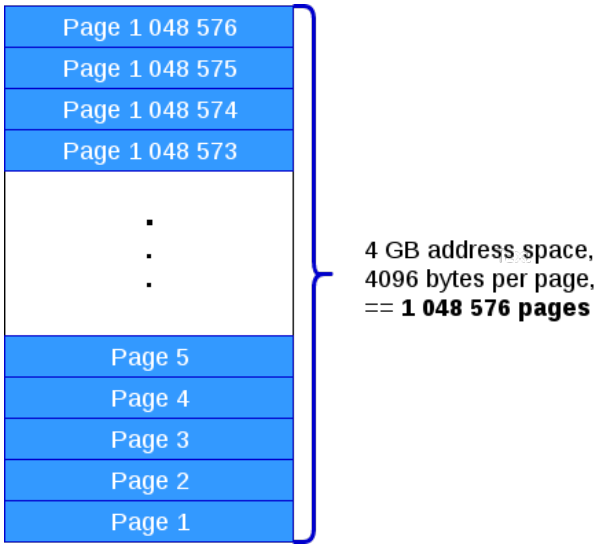

Abbildung 1: Beispiel-Aufteilung von vier Gigabyte Arbeitsspeicher in Pages von 4096 Bytes

Diese Aufteilung hat den Vorteil, dass sie sehr handlich ist: Dem Kernel steht damit eine einheitliche Größe zur Verfügung, mit der gerechnet werden kann.
Neben dem physischen Speicher nutzen heutige Systeme ein Konzept, das ***virtueller Speicher*** genannt wird. Der Name ergibt sich aus der Optik, wo *virtuell* bedeutet, dass Bilder in Spiegeln oder Linsen sichtbar sind, aber nicht wirklich existieren. Für virtuellen Speicher bedeutet das: Der Kernel sichert Prozessen lediglich virtuellen Speicher (der ebenfalls in Pages organisiert wird) zu, der dann benutzt werden kann, als wäre er physisch vorhanden. Das heißt, reserviert ein Prozess Speicher, so wird er einen Verweis auf eine so genannte *Virtual Memory Area (VMA)* erhalten, die eine Art Vertrag zwischen Prozess und Kernel über die Speicherreservierung darstellt. Damit sichert der Kernel zu, dass er die virtuellen Pages auf physischen Speicher mappen wird, sobald der Prozess sie tatsächlich zugreift (zum Beispiel schreibend). Ohne eine solche VMA führt ein Speicherzugriff zu einem *Segmentation fault (Segfault)* und der Prozess wird bestraft, d.h. beendet. Um nachvollziehen zu können, welche virtuelle Seite auf welche physische Seite gemappt wurde, hält jeder Prozess eine *Page Table*, in der zu jeder virtuellen Seite ein *Page Table Entry (PTE)* existiert, in dem unter anderem vermerkt wird, ob die Seite gemapped ist („present“), ob gelesen und/oder geschrieben werden darf (Berechtigungen) und ob der Inhalt sich möglicherweise im Swap befindet[5]. Wichtig ist noch die Feststellung, dass durch diese Konzepte der Zustand jeder Seite (physisch und virtuell) dem Kernel zu jedem Zeitpunkt bekannt ist.
Dass der Kernel den virtuellen Speicher erst bei Bedarf auf physischen Speicher mapped bedeutet im Umkehrschluss, dass er in der Lage ist, mehr Speicher zuzusichern, als tatsächlich verbaut

[4] definiert in `include/asm/page.h`, beispielsweise in *asm-generic*: `http://lxr.free-electrons.com/source/include/asm-generic/page.h#L19`
[5] Befindet sich ein Page-Inhalt im Swap, so beinhaltet sie lediglich eine Adresse, die darauf verweist

ist. Tatsächlich ist dies der Regelfall und wird ***optimistische Speicherverwaltung*** genannt. Der so genannte ***virtuelle Adressbereich*** ist wesentlich größer als der physische: Während es in 32 Bit Systemen die 32-Bit-Grenze (2^{32} Adressen, das sind vier Gigabyte) gab, kann der virtuelle Adressbereich in 64 Bit Systemen bis zu $2^{64} = 18\,446\,744\,073\,709\,551\,616$ Adressen umfassen.

Die Nutzung von virtuellem Speicher und die daraus folgende Möglichkeit, mehr Speicher zu verwalten, als tatsächlich verbaut ist, führt dazu, dass in modernen Systemen nicht ein einzelner, virtueller Adressbereich existiert, sondern vielmehr jeder **Prozess** einen eigenen, virtuellen Adressraum besitzt, den man ***Memory Sandbox*** nennt. Dadurch hat ein Prozess keinen Zugriff auf Pages anderer Prozesse; das ist ein kluger Mechanismus, um die Integrität der Prozesse (und letztlich des Kernels selbst) zu schützen. Darüber hinaus existiert eine weitere Unterteilung des Speichers, die in Abbildung 2 dargestellt wird: In ***User-Space*** und ***Kernel-Space***.

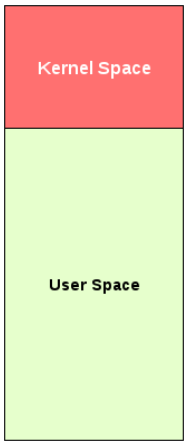

Abbildung 2: User-Space und Kernel-Space

Der Kernel-Space ist identisch für jeden Prozess. Beim *Kontextwechsel*, also dem Wechsel des Prozessors von einem Prozess zu einem anderen, wird nur der User-Space des alten gegen den User-Space des neuen Prozesses ausgetauscht; der Kernel-Space bleibt vorhanden. Er ist ein **priviligierter Speicherbereich** für den Kernel, der vor allem auch den Kernel selbst beinhaltet. Im Kontrast dazu ist der User-Space der Bereich des virtuellen Adressbereichs, dessen Adressen allen Nicht-Kernel-Programmen zur Verfügung stehen: Beispielsweise Firefox, Evince, Thunderbird, aber auch System-Daemons und bestimmte Treiber.

Analog zu User-Space und Kernel-Space existieren noch der ***Kernel Mode*** und der ***User Mode***. Analog zu den Spaces, die aussagen, in welchem Speicherbereich eine Adresse existiert, wird damit bezeichnet, was der Prozess tatsächlich darf. Jeder Space besitzt ein Segment, in dem der eigentliche Code bzw. ein Verweis darauf existiert (zum Beispiel auf `/usr/bin/nano`). Wird Code aus dem Code-Segment des User-Space ausgeführt, so läuft der Prozess selbst im User Mode. Liegt der Code im Kernel-Space, so läuft der Prozess im Kernel Mode. Letzteres wird durch eine Flag auf der *Central Processing Unit (CPU)* gesteuert: `privileged = 1`. Während Prozesse im User Mode nicht viel mehr dürfen, als Daten im User-Space hin- und herzuschieben, darf ein priviligierter Prozess alles (wir erinnern uns: Der System-Kernel hat komplette Kontrolle über das System). Eine Möglichkeit für einen User-Prozess, zum Beispiel Dateien zu öffnen, wofür Kernel-Privilegien benötigt werden, soll hier der Vollständigkeit halber kurz angerissen werden: Der ***Syscall***. Ein Syscall ist ein Aufruf einer Dienstleistung des Kernels (zum Beispiel `open`[6]). Dabei springt die CPU an eine magische Adresse im Kernel-Space, den *Syscall-Entrypoint* und setzt `privileged = 1`. Der Code wird dann in den Kernel-Space kopiert und ausgeführt. Am Ende der Ausführung wird zurückgewechselt und `privileged = 0` gesetzt. Ein non-privileged

[6]Zum Öffnen von Dateien, siehe `http://man7.org/linux/man-pages/man2/open.2.html`

Zugriff vom User Mode auf Kernel-Space-Adressen führt zu einem Segfault, umgekehrt ist der Zugriff gestattet.

Man kann sich leicht vorstellen, dass das Mapping von virtuellem auf physischen Speicher teuer ist. Gerade für den Kernel, der als fundamentaler Bestandteil des Betriebssystems permanent Speicher braucht, um das System lauffähig zu halten, ist das kritisch. Daher ist ein Teil des Arbeitsspeichers **permanent in den Kernel gemapped**. Die zugehörigen ***logischen Adressen*** sind zwar auch virtuell, lassen sich aber über ein *Offset* (eine feste Differenz) auf die physischen Adressen umrechnen. Dadurch wird lediglich das Offset verrechnet. Beispielsweise lässt sich die Adresse 0xC0000000 mit Offset 0x12345678 in die Adresse 0xD2345678 umrechnen. Abbildung 3 illustriert die Speicherbereiche.

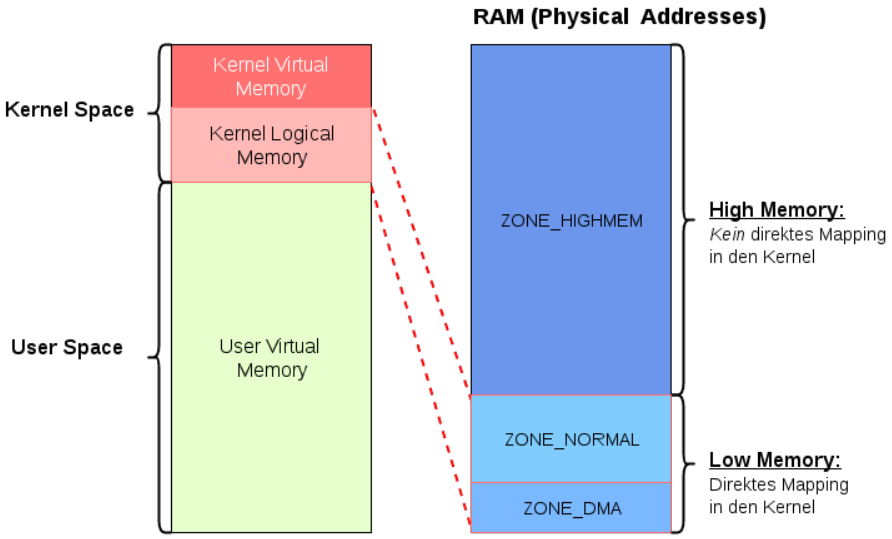

Abbildung 3: Bereiche im virtuellen Adressbereich und im RAM, inspiriert von [9]

Speicher, der permanent in den Kernel gemapped ist, wird ***Low Memory*** genannt, solcher, für den das nicht gilt, ***High Memory***. Obwohl die Graphik etwas anderes suggeriert, ist der Bereich des High Memory in heutigen Systemen weitaus kleiner und nahezu unbedeutend. Er entstammt der Zeit, in der es aufgrund des Vier-Gigabyte-Limits eine Schranke gab, wie viele Adressen der Kernel handhaben konnte.

3 Speicher-Allocation 101

3.1 Arten von Speicher-Allocation

Um Aufgaben erledigen zu können, benötigen Prozesse Arbeitsspeicher. Die Zuteilung von RAM auf die Prozesse übernimmt der Kernel. Dafür muss der Bedarf dem Kernel angemeldet werden; das wird ***Allocation*** genannt[7]. In Abbildung 4 nehmen wir eine noch feinere Aufteilung der Memory Sandbox eines Prozesses vor, indem wir einige wichtige Speichersegmente einzeichnen.

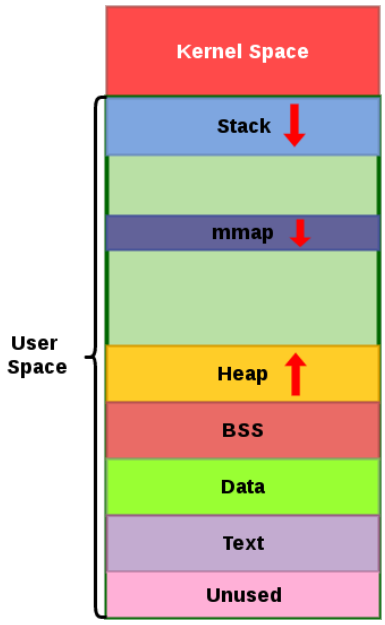

Abbildung 4: Aufteilung des User-Space in Segmente [1]

Diese Segmentierung existiert, in ähnlicher Form, auch im Kernel-Space, ist dort allerdings, trotz identischer Funktion, komplexer aufgebaut; daher konzentrieren wir uns für den Moment auf den User-Space mit dem Hinweis, dass die Angaben übertragbar sind.
Wir haben bereits, ohne die Segmentierung zu kennen, festgestellt, dass der Adressbereich eines Prozesses Code beinhaltet. Dieser liegt im **Text-Segment**. Die anderen Bereiche sind dazu gedacht, abhängig von der Art von Speicherbedarf benutzt zu werden:

- **Mapping von Dateien**: Wird eine komplette Datei im Speicher benötigt, so wird ein Mapping im **mmap-Segment** angelegt.

- **Statische Allocation**: Darunter verstehen wir Speicher, der für statische und globale Variablen benötigt wird: `static int i = 42`. Abhängig davon, ob die Variable initialisiert wurde (wie im Beispiel mit dem Wert 42) oder nicht, liegen sie im **Data-Segment** (initialisiert) bzw. im **BSS-Segment** (nicht initialisiert)[14].

- **Automatische Allocation**: Speicher beispielsweise für Variablen, die im Skopus von Funktionen benötigt werden, wird vom **Stack** reserviert.

- **Dynamische Allocation**: Für statische Allocation ist bei der Kompilierung bereits bekannt, wie viel Speicher benötigt wird; bei der automatischen Allocation ist das ähnlich, wobei der Stack eine initiale Größe hat und nach unten anwächst. Beispielsweise muss der Kernel für ein Programm, das zwei statische Variablen vom Typ int definiert,

[7] vgl. [12]

$2 \times sizeof(int)$ reservieren, nicht mehr, aber auch nicht weniger. Unter dynamischer Allocation verstehen wir Speicherbedarf, der erst zur Laufzeit des Programms zu nicht festgelegten Zeitpunkten und ggf. mit nicht festgelegter Größe entsteht, der daher auch nicht eingeplant werden kann. Dieser Speicher wird vom **Heap** reserviert, wobei auch dieser eine initiale Größe hat und nach oben wächst. Es gibt eine Ausnahme: Ist der Speicherbedarf größer als `MMAP_THRESHOLD`, so wird ein Mapping über `mmap` (Syscall) im entsprechenden Segment angelegt.

Wir stellen fest, dass für die ersten drei Arten der Speicher-Allocation wenige Stellschrauben existieren, um im Kontext von effizienter Programmierung eine Rolle zu spielen. Darunter fällt die Frage, ob ein Wert als Integer, Long, Short oder Float gespeichert werden soll. Jedoch sind diese Entscheidungen keinesfalls so relevant, dass sie einen signifikanten Unterschied machen würden. Im folgenden wird der Fokus daher auf **dynamischer Speicher-Allocation** liegen. Dort stellen sich zum Beispiel folgende Fragen:

- Wie viele Speicherbereiche werden benötigt? Analog: Wie viel soll gespeichert werden?
- Wie groß sollen diese sein? Und, anschließend daran,
- was will ich speichern? Welche Datentypen?
- Kann ich Speicher wiederverwenden? Beispielsweise stellt sich diese Frage, wenn Speicher zu Beginn und zum Ende der Laufzeit einer Funktion benötigt wird.
- etc.

3.2 Speicher-Allocation im User-Space

Im User-Space ist die Funktion, die hauptsächlich zur dynamischen Allocation von Speicher benutzt wird, `malloc`, deren Signatur Listing 2 zeigt.

```
void *malloc(size_t size)
```

Listing 2: Signatur der Funktion `malloc`[8]

Wir stellen fest, dass die Signatur von `malloc` sehr einfach ist: Sie akzeptiert exakt einen Parameter, `size`, der die Größe des zu reservierenden Speicherbereichs in Byte angibt. Im Erfolgsfall liefert `malloc` einen void-Pointer auf das erste Byte eines Speicherbereichs im Heap (oder auf ein mmap-Mapping, sofern $size > MMAP\_THRESHOLD$). Das Ende des Speicherbereichs ergibt sich durch $pointer + size$. Im Misserfolg ist die Adresse 0, das ist der so genannte ***NULL*-Pointer**. Listing 3 zeigt ein Beispiel für die Benutzung.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void main() {
5    int *ptr = malloc(sizeof(int));
6
7    if (ptr == NULL) {
8      exit(EXIT_FAILURE);
9    }
10
11   *ptr = 42;
```

[8] malloc-Spezifikation: [4]

```
12    // do something with ptr...
13    free(ptr);
14  }
```

Listing 3: Ein Beispiel für die Benutzung von `malloc`

In Zeile fünf wird ein Speicherbereich reserviert, der genau groß genug ist, um einen Integer zu speichern. In Zeile elf wird der Wert 42 in den Speicher geschrieben, bevor in Zeile 13 die Funktion `free` den Speicher wieder freigibt[9]. Zeilen sieben bis neun werden in User-Space-Programmen oft weggelassen: Es wird geprüft, ob `malloc` den `NULL`-Pointer zurückgegeben hat, um – in diesem Fall – das Programm zu beenden. Im User-Space hat das Weglassen des Checks kaum Auswirkungen: Im schlimmsten Fall wird der Kernel das Programm beenden. Darüber hinaus nimmt der Kernel dem Entwickler auch sonst viele Aufgaben ab: Ist zum Beispiel nicht mehr genug Speicher im Heap verfügbar, so wird der Kernel ihn vergrößern; wird mehr reserviert, als `MMAP_THRESHOLD`, so wird der Kernel ein Mapping erstellen. Von all diesen Aktionen merkt der Entwickler nichts. Das macht Speicher-Allocation im User-Space sehr einfach. Im Kernel verhält es sich allerdings anders.

4 Speicherverwaltung im Kernel

4.1 Folgen unvorsichtiger Speicher-Allocation User-Space vs. Kernel-Space

Bereits mehrfach angedeutet wurde die Tatsache, dass der Kernel einen Prozess im User Mode beenden wird, sofern beispielsweise auf den `NULL`-Pointer zugegriffen wird oder auf Speicherbereiche, die nicht reserviert wurden. Im Kernel ist das anders. Da der Kernel „alles darf“, sind die Folgen schlimmer:

- Der Prozess kann auch hier beendet werden (Segfault)
- Der Kernel kann andere Prozesse beenden: Benötigt zum Beispiel ein kritischer System-Bestandteil dringend Speicher, so kann der Kernel andere Prozesse beenden, um Speicher freizugeben.
- Der Kernel verliert sich in Mappings und Swappings. Gemeint ist der Umstand, dass der Kernel versuchen muss, eine Speicher-Allocation zu befriedigen. Dafür kann er den Inhalt von Pages in den Swap (auf die Festplatte) schieben, Mappings auf physischen Speicher entfernen, oder auf andere Pages umbiegen etc. In Situationen, in denen wenig Speicher frei ist, kann das dazu führen, dass der Kernel so viele Aktionen ausführen muss, dass er einfach nicht mehr arbeitsfähig ist.
- Es kann eine ***Kernel Panic*** passieren. Das bedeutet, dass der Kernel in einen Zustand gerät, der nicht definiert ist, in dem es also nicht möglich ist, das System sicher weiterzubetreiben, weswegen er sich in der Folge selbst abschaltet[15].
- Aus den vorherigen Punkten können **Datenverlust** oder sogar die **Unbenutzbarkeit des Systems** folgen.

Es ist also im Kernel, im Vergleich zum User-Space, größere Vorsicht geboten.

4.2 Speicher-Allocation im Kernel-Space

Im Kernel existieren vor allem zwei Funktionen, die bedeutend für die Reservierung von Speicher sind:

[9] free-Spezifikation: [3]

- Das ist einmal die Funktion `vmalloc`, deren Funktionsweise der von `malloc` sehr ähnlich ist: Beide Funktionen geben Speicher zurück, der **virtuell zusammenhängendend** ist.
- Die zweite, wichtige Funktion, ist `kmalloc`. Speicher, der mit `kmalloc` alloziert wird, ist **physisch und virtuell zusammenhängend** und entstammt dem **logischen Adressbereich**. Dabei ist der physische Zusammenhang ein **Muss**, wie wir im Folgenden sehen werden.
- Der Vollständigkeit halber seien noch `kfree`[10] und `vfree`[11] erwähnt, die, wie `free` im User-Space, den Speicher wieder freigeben. Als Eselsbrücke: `vfree` gibt Speicher von `vmalloc` wieder frei; `kfree` von `kmalloc`.

4.3 kmalloc

```
void *kmalloc(size_t size, int flags)
```

Listing 4: Signatur der Funktion `kmalloc`[12]

Listing 4 zeigt die Signatur von `kmalloc`. Im Vergleich zu `malloc` fällt auf, dass die Funktion zwei Parameter akzeptiert: Neben der bekannten Größe **size** noch **flags**. Während size angibt, **wie viele Pages** benötigt werden, steuert flags, **wie Pages alloziert werden**. Dabei handelt es sich um einen Integer, deren Wert sich durch das Bitwise-Oder verschiedener Konstanten ergibt. Die einzelnen Werte sind in `include/linux/gfp.h`[13] definiert, wobei `slab.h`, in der `kmalloc` definiert wird, `gfp.h` einbindet. Wichtige Flags sind:

- **__GFP_WAIT**: Damit wird angegeben, dass der Prozess auf freie Pages **warten** kann; das bedeutet, der Kernel kann den Prozess blockieren, bis Seiten frei werden.
- **__GFP_HIGHMEM**: Der Kernel darf high memory zugreifen, um den Request zu befriedigen.
- **__GFP_DMA**: Die Zone *Direct Memory Access Segment (DMA)* kann zugegriffen werden.
- **__GFP_ZERO**: Die Seite wird mit Nullen gefüllt, bevor sie dem Prozess zur Verfügung gestellt wird.
- **__GFP_HIGH**: Der Kernel hält einen Speicherbereich immer frei, um ihn für Notfälle nutzen zu können. Diese Flag gibt an, dass dieser Bereich angezapft werden darf.
- **__GFP_FS**: Der Kernrel darf Operationen im Dateisystem ausführen, um Speicher freizuräumen (Swapping)
- ...
- **GFP_KERNEL**: Entspricht __GFP_WAIT |__GFP_IO |__GFP_FS und wird für Speicherbedarf im Kernel ohne besondere Anforderungen genutzt.
- **GFP_USER**: Wie GFP_KERNEL für den User-Space
- **GFP_ATOMIC**: Entspricht __GFP_HIGH und wird in atomaren Kontexten genutzt, wo dringend Speicher benötigt wird.

[10] kfree-Spezifikation: [5]
[11] vfree-Spezifikation: [7]
[12] kmalloc-Spezifikation: [6]
[13] siehe `http://lxr.free-electrons.com/source/include/linux/gfp.h`

- **GFP_REPEAT**
  GFP_NOFAIL
  GFP_NORETRY
  Diese Flags steuern, was passieren soll, wenn der Speicherbedarf nicht gedeckt werden kann. Erstere sagt aus, dass der Kernel wiederholt versuchen soll, freie Seiten zu finden. Die anderen beiden stehen sich gegenüber: NOFAIL bedeutet, dass die Allocation derart systemrelevant ist, dass sie nicht scheitern darf. NORETRY meint das Gegenteil: Der Kernel soll nach einem Versuch aufgeben.

In Listing 5 sehen wir eine beispielhafte Benutzung von `kmalloc` in einem Kernel-Modul.

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/kernel.h>
4  #include <linux/slab.h> /* kmalloc, kfree */
5
6  static int __init kmalloc_test_init(void) {
7    int *ptr = kmalloc(sizeof(int), GFP_KERNEL);
8
9    if (ptr == NULL) {
10     return -1;
11   }
12
13   *ptr = 42;
14   // do something with ptr...
15   printk(KERN_INFO "ptr is %d", *ptr);
16   kfree(ptr);
17
18   return 0;
19 }
```

Listing 5: Beispiel für die Verwendung von `kmalloc`

Ähnlich wie im `malloc`-Beispiel in Listing 3 wollen wir einen Integer speichern, um ihn in Zeile 15 in eine Log-Datei (üblich etwa `/var/log/kern.log`) zu schreiben. Benutzt wird `GFP_KERNEL`, um eine reguläre Allocation von Kernel-Speicher zu signalisieren. Wir haben bereits festgestellt, dass der `NULL`-Check in den Zeilen neun bis elf im Kernel äußerst wichtig ist: Trotz `GFP_NOFAIL` kann trotzdem der `NULL`-Pointer zurückgegeben werden.

Wichtig ist die Feststellung, dass der Flags-Parameter zwar große Freiheit in dem Sinne bietet, als dass das Verhalten von `kmalloc` je nach Verwendungszweck angepasst werden kann, mit ihm aber auch Risiken einhergehen. Einige Beispiele:

- Wird ein Zonen-Modifikator wie __GFP_DMA genutzt, kann `NULL` zurückgegeben werden, weil dort zu wenig Speicher frei ist, obwohl zum Beispiel im low memory genug vorhanden wäre.

- Mit __GFP_WAIT kann das Programm blockieren. Dies kann unter Umständen unerwünscht sein. Oder umgekehrt: Die Flag wird nicht gesetzt und führt in Kombination dazu, dass der Request nicht befriedigt werden kann.

- Es kann zu einer **Endlosrekursion** führen. Angenommen, es wird Code für einen Teil des Dateisystems geschrieben, die Flag GFP_NOFS aber nicht gesetzt, so kann die Allocation zu Operationen am Dateisystem führen, wodurch der betroffene Teil Speicher braucht, der wieder zu Operationen am Dateisystem führt usw.; dies resultiert in einem Stackoverflow und ggf. einer Kernel Panic oder sogar Datenverlust.

- Es können auch **Deadlocks** auftreten: Zum Decken des Speicherbedarfs soll geswapped werden, der Treiber benötigt dafür aber Speicher

Neben dem Flags-Parameter ist eine zweite Eigenschaft von `kmalloc` wichtig: Es gibt **physisch zusammenhängenden Speicher** zurück. Interessant ist das zum Beispiel für Gerätetreiber, denn, wie man sich leicht vorstellen kann, ist es günstiger, in benachbarten Regionen des Speichers zu arbeiten, als in solchen, die verteilt sind. Dass die Rückgabe von physisch zusammenhängendem Speicher nicht nur ein Vorteil ist, illustriert Abbildung 5.

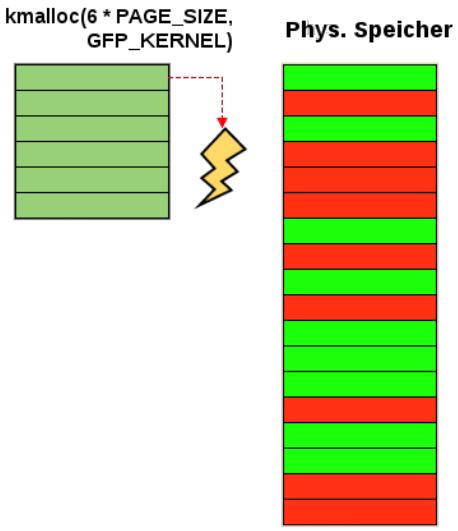

Abbildung 5: kmalloc: Versuch, sechs Pages zu bekommen

Grüne Bereiche signalisieren freie Seiten, rote Bereiche sind belegt. Der Versuch, sechs freie Pages zu bekommen, schlägt hier fehl: Es sind zwar mehr als sechs Seiten im physischen Speicher frei, aber sie sind nicht zusammenhängend.
Wie wir wissen, ist physischer Speicher begrenzt. Daher existiert ein **Limit** für den size-Parameter, der architekturabhängig ist. Soll der Code portierbar sein (also auf verschiedenen Architekturen funktionieren), so darf mit nicht mehr als **128 Kilobyte** gerechnet werden.

4.4 vmalloc

```
void *vmalloc(unsigned long size)
```

Listing 6: Signatur der Funktion `vmalloc`[14]

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/kernel.h>
4  #include <linux/vmalloc.h> /* vmalloc, vfree */
5
6  static int __init vmalloc_test_init(void) {
7    int *ptr = vmalloc(sizeof(int));
8
9    if (ptr == NULL) {
10     return -1;
```

[14] vmalloc-Spezifikation: [8]

```
11    }
12
13    *ptr = 42;
14    // do something with ptr...
15    printk(KERN_INFO "ptr is %d", *ptr);
16    vfree(ptr);
17
18    return 0;
19  }
```

Listing 7: Beispiel für die Verwendung von `vmalloc`

Listing 6 zeigt die Signatur von `vmalloc` und Listing 7 ein Beispiel. Im Kontrast zu `malloc` und `kmalloc` hat der size-Parameter einen anderen Datentyp: **unsigned long**, dessen Wertebereich in C bis 4 294 967 295 reicht[15], das heißt, es kann wesentlich mehr Speicher alloziert werden. Das hat vor allem damit zu tun, dass `vmalloc` **virtuell zusammenhängenden Speicher** zurückgibt, der im Vergleich zu `kmalloc` zwar physisch zusammenhängen **kann**, aber nicht **muss**. In Abbildung 5 haben wir gesehen, dass `kmalloc` scheitern kann, obwohl genug Pages frei sind, weil der Speicher physisch zusammenhängen muss. Abbildung 6 zeigt dieselbe Situation für `vmalloc`.

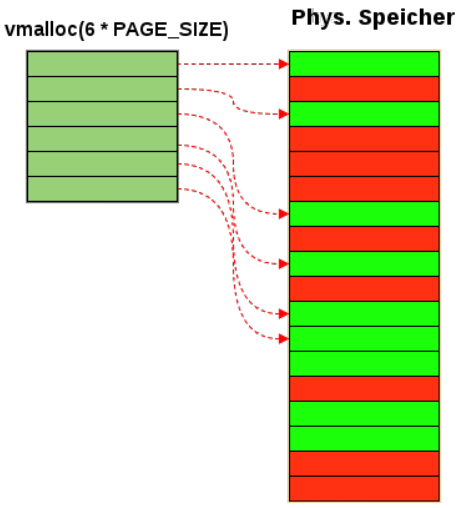

Abbildung 6: vmalloc: Versuch, sechs Pages zu bekommen

`vmalloc` kann trotzdem Pages allocieren, indem es verteilte, physische Seiten in einen virtuellen Block zusammenfasst. Dadurch hat es eine deutlich höhere Erfolgswahrscheinlichkeit.

Tatsächlich ist es so, dass Code, der `vmalloc` nutzt, beim Einbringen in den Linux-Kernel die kalte Schulter gezeigt bekommt[16] (Notiz: Mindestens einer der Autoren der Quelle ist ein wichtiger Maintainer des Kernels). Warum ist das so? Warum nutze ich nicht immer `vmalloc`, wenn es so hohe Erfolgswahrscheinlichkeiten garantiert? Das Problem ist, dass der virtuelle Speicher erst auf physischen Speicher gemapped werden muss. Das bedeutet, zuerst einmal muss der Kernel freie, physische Pages finden. Danach müssen diverse Instanzen aktualisiert werden. Existiert für den Prozess noch keine Page Table, so muss diese angelegt inklusive aller PTE angelegt werden. Dafür wird wiederum Speicher benötigt. Ein wichtiger Punkt ist, dass der Kernel für das Allocieren des Speichers für die Page Tables `kmalloc` mit GFP_KERNEL nutzt, woraus resultiert, dass der komplette Prozess blockieren kann. Dadurch kann `vmalloc` nicht in einem atomaren Kontext genutzt werden. Insgesamt ist `vmalloc` also vor allem dann sehr teuer, wenn

[15] vgl. [10, p. 23]
[16] vgl. [2, p. 225]

wenige Pages benötigt werden. Aber auch für große Mengen ist es, aufgrund des Mappings, nicht günstig.

4.5 Effiziente Speicherverwaltung im Kernel

Die Benutzung von sowohl `kmalloc` als auch `vmalloc` bringt Vor- und Nachteile. Es wäre daher nicht angemessen, eine generelle Empfehlung für oder wider einer Funktion auszusprechen. Wichtig ist, dass im Kernel **effizient** vorgegangen wird, das heißt der Entwickler sollte lieber einmal mehr nachdenken. In letzter Konsequenz kann es notwendig sein, das Programm an sich zu überdenken. Im Folgenden dazu einige Beispiele. Dazu soll angenommen werden, dass ein `void *data` existiert, in dem 100 Megabyte Daten referenziert werden – woher diese Daten kommen, ist dabei egal. Gegeben sei außerdem die Funktion, deren Signatur in Listing 8 beschrieben wird und die Daten von src komprimiert und nach dest schreibt.

```
void compress(void *src, void *dest);
```

Listing 8: Signatur einer Kompressions-Funktion

4.5.1 Ein sehr schlechtes Beispiel

Im User-Space würde man vielleicht einfach vorgehen wie in Listing 9.

```
1  static int __init compressor_init(void) {
2    void *destination = kmalloc(100M, GFP_KERNEL);
3
4    if (destination == NULL) {
5      return -1;
6    }
7
8    compress(data, destination);
9    kfree(destination);
10 }
```

Listing 9: Sehr schlecht: Allocation von 100 MB im Kernel

Versucht man, dieses Modul in den Kernel zu laden, wie in Listing 10, so wird ein Fehler geworfen. Tatsächlich ist das Beispiel blauäugig: Im Kernel wird es niemals passieren, dass 100 Megabyte Speicher auf einmal reserviert werden, weder mit `kmalloc`, noch mit `vmalloc`.

```
# insmod example1.ko
insmod: ERROR: could not insert module example1.ko: Operation not
    permitted
```

Listing 10: Der Versuch, das Beispiel in den Kernel zu laden, wird sofort zurückgewiesen

4.5.2 Ein etwas besseres Beispiel

Das Beispiel in Listing 11 ist etwas besser: Der Entwickler versucht, die Daten in handliche Teile von `PAGE_SIZE` zu zerlegen, jede Seite einzeln zu komprimieren und am Ende zusammenzuführen.

```
1  static int __init compressor_init(void) {
```

```
2   void *destination; char* result;
3
4   for (each part of data of size PAGE_SIZE) {
5     destination = vmalloc(PAGE_SIZE);
6
7     // Nullcheck...
8
9     compress(current part of data, destination);
10    vfree(destination);
11    // do stuff to merge destination into result
12  }
13
14  return 0;
15 }
```

Listing 11: Immer noch schlecht: Allocation in einer for-Schleife

Allerdings alloziert er in jedem Schleifendurchlauf Speicher und gibt ihn danach wieder frei. Scheitert ein Allocations-Versuch, kann unter Umständen die komplette Kompression fehlschlagen. Im Beispiel wurde `vmalloc` verwendet, ist hier jedoch problemlos gegen `kmalloc` austauschbar.

4.5.3 Ein gutes(?) Beispiel

Das Beispiel in Listing 12 korrigiert die Fehler der beiden vorangegangenen Beispiele: Jetzt wird einmalig ein Speicherbereich alloziert und wiederverwendet.

```
1  static int __init compressor_init(void) {
2    void *workmem = vmalloc(PAGE_SIZE); char *result;
3
4    // Nullcheck...
5
6    for (each part of data of size PAGE_SIZE) {
7      compress(current part of data, workmem);
8      // do stuff to merge workmem into result
9    }
10
11   vfree(workmem);
12
13   return 0;
14 }
```

Listing 12: Einmalige Allocation eines working memory und Wiederverwendung, optimal(?)

Ob dieses Beispiel optimal ist, soll hier nicht definitiv beantwortet werden, da die Beispiele lediglich ein Gefühl vermitteln sollten, wie Programme im Kernel arbeiten könnten.

4.6 Weitere Beispiele

Es ist Aufgabe des Entwicklers, für seinen Anwendungsfall bestimmte Entscheidungen zu treffen. Ist es zum Beispiel wichtig, physisch zusammenhängenden Speicher zu bekommen? Wie viel Speicher brauche ich? Sind ein großer Bereich notwendig – oder mehrere kleine? Tun mir die Extra-Kosten von `vmalloc` weh im Vergleich zu seinen Vorteilen? Welche Auswirkungen hat es, wenn ich keinen Speicher bekomme? Welche Art von Speicher will ich? In den Listings 13, 14 und 15 sehen wir weitere Beispiele dazu.

```
1
```

```
void *workmem = kmalloc(PAGE_SIZE, GFP_KERNEL);

for (each part of data of size PAGE_SIZE) {
  compress(current part of data, workmem);
}
```

Listing 13: Das vorherige Beispiel mit `kmalloc` statt `vmalloc`

```
void *workmem = kmalloc(3 * PAGE_SIZE, GFP_KERNEL);

for (each part of data of size 3 * PAGE_SIZE) {
  compress(current part of data, workmem);
}
```

Listing 14: Bearbeiten von drei Pages zur Zeit statt nur einer

```
void *workmem = vmalloc(10 * PAGE_SIZE);

for (each part of data of size 10 * PAGE_SIZE) {
  compress(current part of data, workmem);
}
```

Listing 15: Doch wieder `vmalloc`, dafür mehr Seiten

5 Fazit

Wir haben gesehen, dass Speicherverwaltung im User-Space ziemlich einfach ist: Der Kernel nimmt dem Entwickler den Großteil der Denkarbeit ab. Wird allerdings für den Kernel selbst programmiert, so stellen sich plötzlich Fragen, die im User-Space keine Rolle spielen, deren Antworten allerdings den Unterschied zwischen einem effizienten, einem ineffizienten oder einem gar nicht erst lauffähigen Programm machen. Und es treten Probleme auf, die es zu lösen gilt. Insgesamt ist es jedoch notwendig, da gerade der Kernel effizient arbeiten muss, damit die User-Space-Programme „entspannt“ laufen können.

Literaturverzeichnis

[1] Vincent Bernat. URL: https://d1g3mdmxf8zbo9.cloudfront.net/images/memory-layout-regular.png.

[2] Jonathan Corbet, Alessandro Rubini und Greg Kroah-Hartmann. *Linux device drivers*. third. O'Reilly und Associates, Feb. 2005.

[3] The IEEE und The Open Group. *free*. 2016. URL: hhttp://pubs.opengroup.org/onlinepubs/009695399/functions/free.html (abgerufen am 20. 01. 2017).

[4] The IEEE und The Open Group. *malloc*. 2016. URL: http://pubs.opengroup.org/onlinepubs/009695399/functions/malloc.html (abgerufen am 16. 12. 2016).

[5] The Linux Kernel organization. *kfree*. URL: https://www.kernel.org/doc/htmldocs/kernel-api/API-kfree.html (abgerufen am 16. 12. 2016).

[6] The Linux Kernel organization. *kmalloc*. URL: https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html (abgerufen am 16. 12. 2016).

[7] The Linux Kernel organization. *vfree*. URL: https://www.kernel.org/doc/htmldocs/kernel-api/API-vfree.html (abgerufen am 16. 12. 2016).

[8] The Linux Kernel organization. *vmalloc*. URL: https://www.kernel.org/doc/htmldocs/kernel-api/API-vmalloc.html (abgerufen am 16. 12. 2016).

[9] Alan Ott. *Virtual Memory and Linux*. Apr. 2016. URL: https://events.linuxfoundation.org/sites/events/files/slides/elc_2016_mem.pdf.

[10] Peter Prinz und Tony Crawford. *C in a nutshell*. Sebastopol, CA: O'Reilly und Associates, 2005.

[11] The Linux Information Project. *Kernel*. Mai 2005. URL: http://www.linfo.org/kernel.html.

[12] technopedia. *Memory Allocation*. URL: https://www.techopedia.com/definition/27492/memory-allocation (abgerufen am 27. 12. 2016).

[13] Wikipedia. *Arbeitsspeicher — Wikipedia, Die freie Enzyklopädie*. 2017. URL: https://de.wikipedia.org/w/index.php?title=Arbeitsspeicher&oldid=161758406 (abgerufen am 28. 01. 2017).

[14] Wikipedia. *Data segment — Wikipedia, The Free Encyclopedia*. 2016. URL: https://en.wikipedia.org/w/index.php?title=Data_segment&oldid=755754424 (abgerufen am 19. 01. 2017).

[15] Wikipedia. *Kernel panic — Wikipedia, The Free Encyclopedia*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Kernel_panic&oldid=762283000 (abgerufen am 28. 01. 2017).

Abbildungsverzeichnis

Listing-Verzeichnis

Anhang

A Benutzte Konventionen

Bold
: Wird benutzt, um wichtige **Schlüsselbegriffe** zu kennzeichnen.

Italic
: Wird benutzt, um Begriffe zu kennzeichnen, die vorher noch nicht verwendet wurden und die im jeweiligen Kontext definiert werden.

Italic bold
: Wird benutzt, um Begriffe zu kennzeichnen, die vorher noch nicht verwendet wurden und die wichtige Schlüsselbegriffe darstellen.

`Constant Width`
: Wird für Funktionsnamen benutzt (`malloc`), aber auch für Konstanten (`PAGE_SIZE`), Shell-Befehle (`getconf`) oder Pfade (`/usr/bin/nano`).

$, #
: Werden in den Beispiel-Eingaben in die Shell benutzt, um Eingaben als Root (#) bzw. User ($) zu verdeutlichen:

```
$ sudo su
# ...
```

B Abkürzungsverzeichnis

CPU Central Processing Unit. 5

DMA Direct Memory Access Segment. 10

I/O Input/Output. 3

OS Operating System. 3

PTE Page Table Entry. 4, 13

RAM Random Access Memory. 3, 4, 7

Segfault Segmentation fault. 4, 6, 9

VMA Virtual Memory Area. 4