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

Speicherverwaltung im Kernel

Sven Schmidt am 26. Januar 2017

Bild: [Zah14]

Motivation: Speicher ist begrenzt

- Programme brauchen Arbeitsspeicher, um zu arbeiten
- Speichern von Werten, Tracking von Zuständen, Mapping von Dateien
- Tendenz zum Speicher-Hunger
- Aber: Speicher ist begrenzt
- Speicher muss effizient verwaltet werden
- Einige Sprachen: Manuelle Speicherverwaltung (C, Sprache des Linux-Kernels)

Gliederung

Speicher

- **Speicher** meint Arbeitsspeicher (RAM)
- Zweck: Bereithalten von Teilen des OS, von Programmen oder Daten, die gerade jetzt oder häufig genutzt werden
- Viel höhere Zugriffsgeschwindigkeiten als Speichergeräte (Festplatten)
- Günstig, dafür in der Kapazität beschränkt

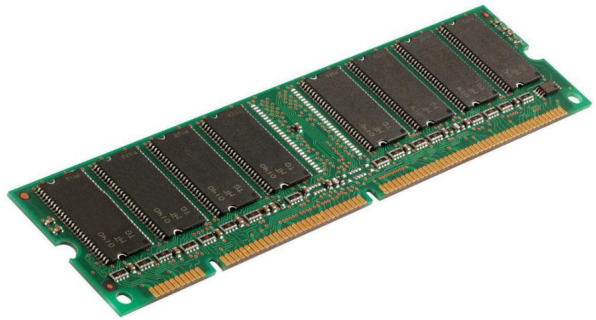

Abbildung: Ein RAM-Riegel [dir]

Kernel

1. Zentraler **Kern** eines Betriebssystems
2. **Komplette Kontrolle** über das gesamte System
3. Erste Komponente, die beim Boot in den Speicher geladen wird; verbleibt dort während der Laufzeit
4. Bietet fundamentale Dienstleistungen
   - Gerätetreiber und -verwaltung
   - Prozessverwaltung und -scheduling
   - **Speicherverwaltung**
   - Dateimanagement
   - I/O-Verwaltung (Zugriff auf Peripherie)
5. Das eigentliche **Betriebssystem** baut auf diesen Services auf und bietet zusätzlich Programme wie eine Oberfläche
6. Dieser Vortrag: **Linux-Kernel**

Bild: [ker], vgl. [Pro05]

Organisation des Speichers durch den Kernel

- Der Kernel organisiert Speicher in **Pages** von `PAGE_SIZE`
  - Üblich z.B. 4.096 bytes
  - Handlich: Gleich große Einheiten zum rechnen
- **Adresse**: Verweis auf den Beginn eines Speicherbereich (Ende kennen wir, warum?)
- **Pointer**: *Zeigt* auf Adresse; kann hin- und hergereicht werden
- **Adressbereich**: Adressierbarer Speicherbereich

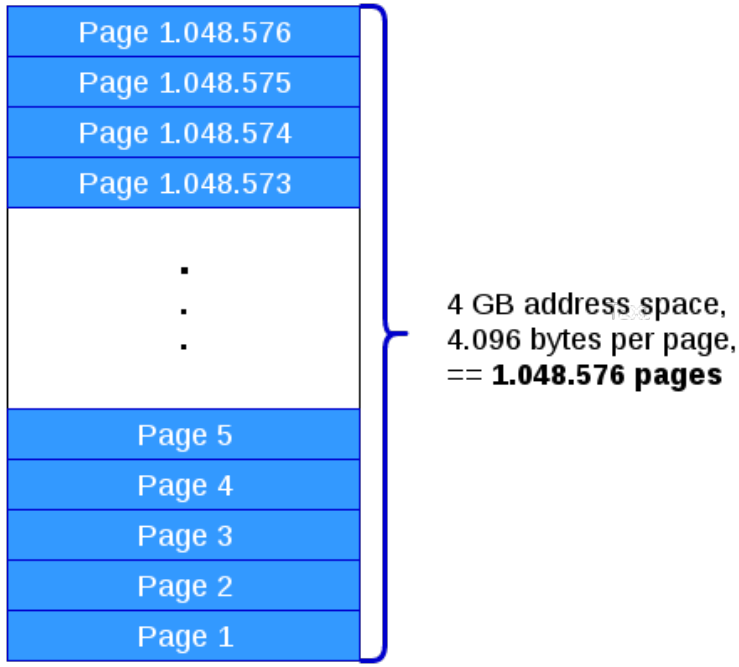

Abbildung: Beispiel-Aufteilung in Pages

PAGE_SIZE ist z.B. definiert in page.h: http://lxr.free-electrons.com/source/include/asm-generic/page.h#L17

Prozess-Speicher-Layout

- Jeder **Prozess** besitzt eigenen **Adressbereich** (*Memory Sandbox*)
- Kein Zugriff auf Speicher anderer Prozesse (Schutzmechanismus)
- Der Adressbereich besteht aus **virtuellem Speicher** (dazu gleich)
- Ein Teil heißt **Kernel-Space**
  - Identisch für jeden Prozess
  - Priviligierter Speicher-Bereich für den Kernel
- Außerdem: **User-Space**
  - Privat für den Prozess
  - Austausch beim **Kontextwechsel**

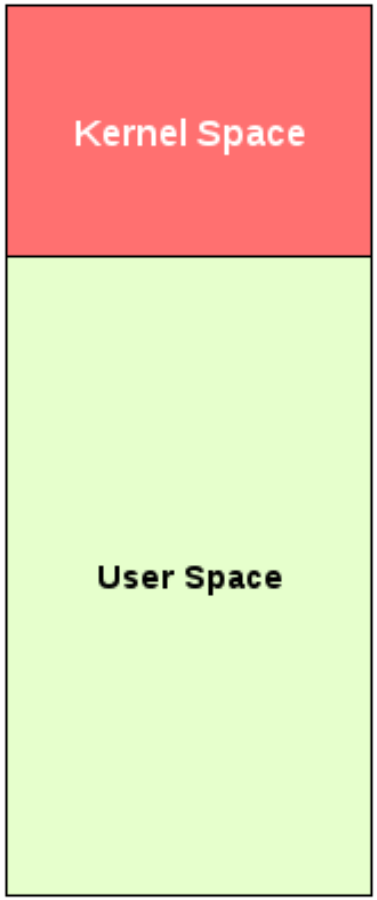

Abbildung: User-Space und Kernel-Space

Virtueller Speicher: Eigenschaften

- Name aus der **Optik**: Bilder in Spiegeln und Linsen, die nicht wirklich da sind $\Rightarrow$ Speicher, der nicht wirklich da ist
- Kann benutzt werden, als wäre er physisch vorhanden
  - Kernel: **Mapping auf physischen Speicher** bei Nutzung
  - Der Kernel ist bereit, mehr virt. Speicher zuzusichern, als physisch verbaut (**optimistische Speicherverwaltung**)
- Virtueller Adressbereich wesentlich größer als physischer
  - 32 Bit: 2^{32} Adressen: 1GB Kernel-Space, 3GB User-Space
  - 64 Bit: 2^{64} Adressen: Aufteilung unterschiedlich
- Funktionen geben i.d.R. Pointer auf virtuelle Adressen zurück (egal, ob Kernel- oder User-Space)

User Mode und Kernel Mode

- Beim Start eines Prozesses
  - Ausführung von Code aus User- oder Kernel-Space
  - Je nachdem: Prozess läuft im **Kernel Mode** oder im **User Mode** (`privileged`-Flag auf der CPU)
  - Kernel Mode: Darf annähernd alles
- Ausführen von Operationen in User-Space-Code, die priveleged = true erfordern (Öffnen von Dateien)
  - Anrufen des Kernels (s.g. **Syscall**)
  - CPU springt an **magische Adressen** im Kernel Space
  - Priveleged = true
  - Der Code wird in den Kernel-Space kopiert und ausgeführt
- Es existieren noch andere Möglichkeiten
- non-privileged Zugriff auf Kernel-Space: **Segmentation fault**

Arten von Speicher(-adressen)

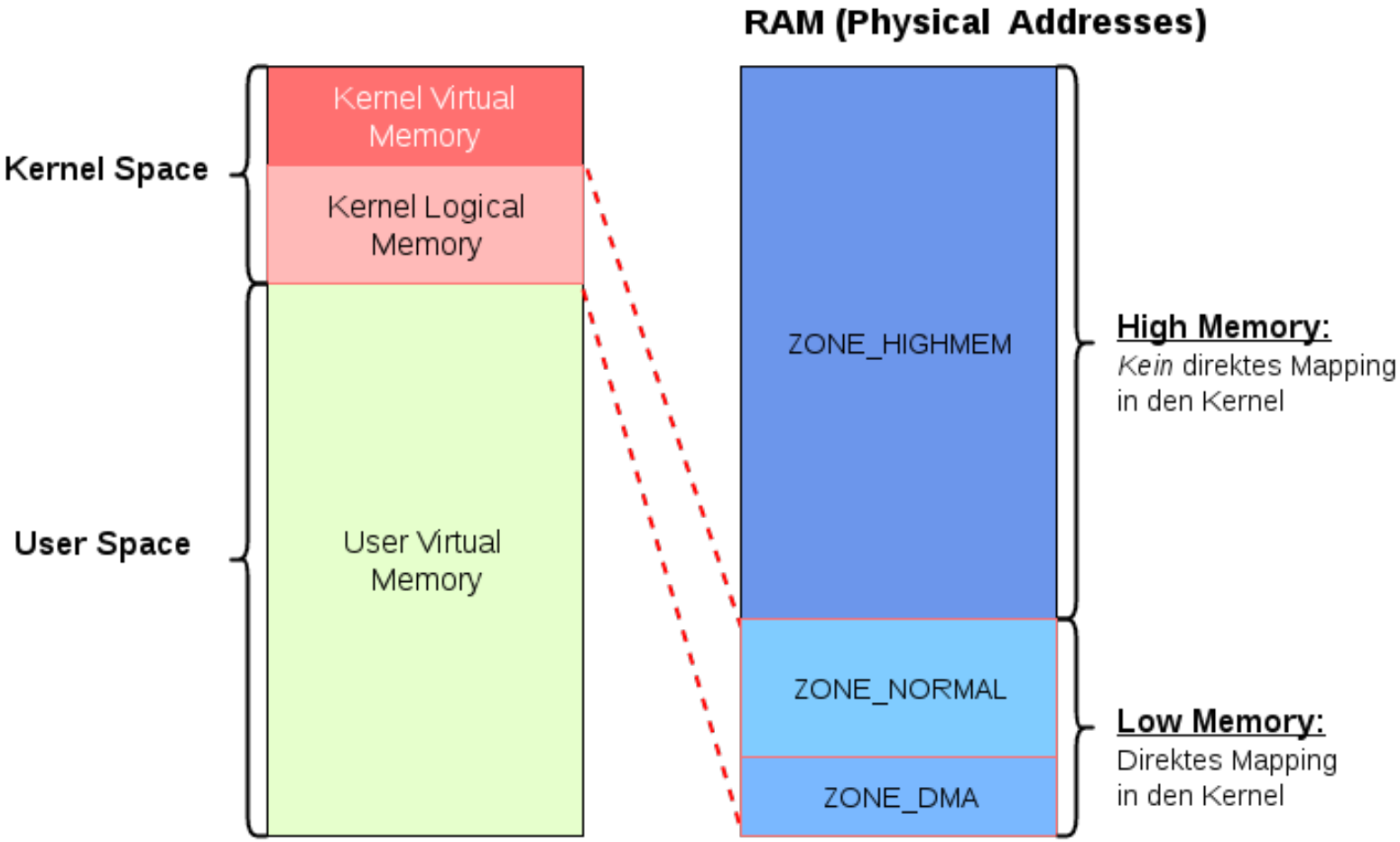

Abbildung: Adress-Bereiche und RAM, inspiriert von [Ott16]

Arten von Speicher(-adressen)

- Physical memory
  - Tatsächlich verbauter Speicher
- (Kernel) logical memory
  - 1:1-Mapping von physischem Speicher auf den logischen Speicher im Kernel
  - Kann nicht in den Swap verschoben werden
  - Adressen sind virtuell, lassen sich aber über ein Offset umrechnen
- (Kernel) virtual memory
  - Speicher ohne direktes Mapping
  - Mapping von virtuellen Adressen auf physischen Speicher bei Bedarf durch den Kernel
  - Adresse des physischen Speichers nicht berechenbar

Memory allocation

"Memory allocation is the process of reserving a partial or complete portion of computer memory for the execution of programs and processes."

— Definition Memory Allocation von [tec]

Drei Arten von Memory allocation

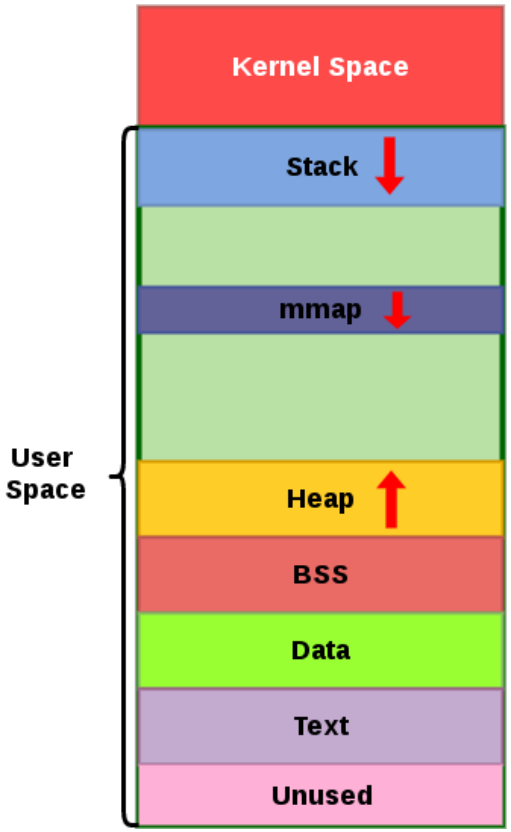

Abbildung: Aufteilung des User-Space in Segmente [Ber]

1. **Statisch** (BSS/Data): *Global* und *static* (initialisiert o. nicht)
2. **Automatisch** (Stack):

```
void foo() {
  // reserviert sizeof(int)
  int i = 10;
}
```

3. **Dynamisch** (Runtime)
   - Heap
   - mmap (komplette Dateien oder Devices)

Vgl. [Fer14]

Effiziente Speicherverwaltung

- Wenn wir von „effizienter Speicherverwaltung“ sprechen, meinen wir **dynamische allocation**
- Stack- oder statische allocation: Kaum Stellschrauben
  - „Heute nutzen wir mal weniger `ints`“
  - long vs. int vs. short
- Dynamische allocation
  - Wie viele Speicherbereiche brauche ich?
  - Wie groß sollen sie sein?
  - Was will ich speichern?
  - Kann ich Speicher wiederverwenden?
  - etc.

Dynamische Speicher-Allocation im User-Space

- `void *malloc(size_t size)`
- *size* ist (fast) nicht beschränkt
- Liefert einen Pointer auf die Startadresse eines Speicher-Bereichs zurück
  - Normalerweise vom Heap
  - Mehr als `MMAP_THRESHOLD`: Erstellen eines anonymen Mappings (**mmap-Syscall**, Kernel)
- Misserfolg: `NULL`-Pointer
- Ist der Heap zu klein, wird er vergrößert (Kernel)
- Der Speicher-Bereich ist virtuell, Mapping auf physischen Speicher erst bei Nutzung (Kernel)
- Freigeben von Speicher: `void free(void *ptr)`

Spezifikation: [IG16]

Ein Beispiel

```
1  void main() {
2    int *ptr = malloc(sizeof(int));
3
4    if (ptr == NULL) {
5      exit(EXIT_FAILURE);
6    }
7
8    *ptr = 42;
9    // do stuff with ptr...
10   free(ptr);
11 }
```

Ein Beispiel

```
1  void main() {
2    int *ptr = malloc(sizeof(int));
3
4    if (ptr == NULL) {
5      exit(EXIT_FAILURE);
6    }
7
8    *ptr = 42;
9    // do stuff with ptr...
10   free(ptr);
11 }
```

- Der Check auf den NULL-Pointer (4 - 6) wird oft weggelassen; im User-Space hat das kaum Auswirkungen

Ein Beispiel

```
1  void main() {
2    int *ptr = malloc(sizeof(int));
3
4    if (ptr == NULL) {
5      exit(EXIT_FAILURE);
6    }
7
8    *ptr = 42;
9    // do stuff with ptr...
10   free(ptr);
11 }
```

- Der Check auf den NULL-Pointer (4 - 6) wird oft weggelassen; im User-Space hat das kaum Auswirkungen
- Das „Mitdenken“ (mmap, Heap vergrößern, Mapping) übernimmt der Kernel $\Rightarrow$ alles sehr flauschig

Ein Beispiel

```
1  void main() {
2    int *ptr = malloc(sizeof(int));
3
4    if (ptr == NULL) {
5      exit(EXIT_FAILURE);
6    }
7
8    *ptr = 42;
9    // do stuff with ptr...
10   free(ptr);
11 }
```

- Der Check auf den NULL-Pointer (4 - 6) wird oft weggelassen; im User-Space hat das kaum Auswirkungen
- Das „Mitdenken“ (mmap, Heap vergrößern, Mapping) übernimmt der Kernel ⇒ alles sehr flauschig
- Im Kernel ist alles nicht so einfach...

Im Kernel ist vorsicht geboten

Was soll denn schon passieren?

- User-Space: Kernel *killt* Prozess
- Kernel-Space:
  - Segfault
  - Kernel killt andere Prozesse
  - Kernel *verliert* sich in Mappings/Swapping, Leeren von Buffern etc.
  - Kernel panic
  - Datenverlust
  - Unbenutzbarkeit des Systems

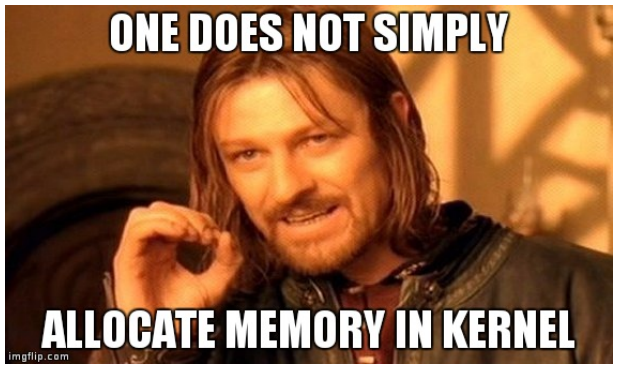

Abbildung: [mem]

Dynamische Speicher-Allocation im Kernel-Space

- `void *vmalloc(unsigned long size)`
  - Gibt **virtuell** zusammenhängenden Speicher zurück
  - Entspricht am ehesten **malloc**
  - Speicher aus dem **virtuellen Adressbereich**
- `void *kmalloc(size_t size, int flags)`
  - Gibt **physisch und virtuell** zusammenhängenden Speicher zurück
  - Speicher aus dem **logischen Adressbereich**
  - Kann dadurch nicht in den Swap verschoben werden
  - Das Verhalten wird durch `int flags` gesteuert
- Freigeben: `void kfree(void *ptr)` bzw. `void vfree(void *ptr)`

---

kmalloc: [orgb], vmalloc: [orgd], kfree: [orga], vfree: [orgc]

Beispiel: kmalloc

```
1  static int __init kmalloc_test_init(void) {
2    int *ptr = kmalloc(sizeof(int), GFP_KERNEL);
3
4    if (ptr == NULL) {
5      return -1;
6    }
7
8    *ptr = 42;
9    // do stuff with ptr...
10   printk(KERN_INFO "ptr is %d", *ptr);
11   kfree(ptr);
12
13   return 0;
14 }
```

kmalloc: Der Flags-Parameter

- `void *kmalloc(size_t size, int flags)`
- Beeinflussen das Verhalten, *wie* freie Pages reserviert werden (GFP = **G**et **F**ree **P**ages)
  - __GFP_WAIT: Kann warten
  - __GFP_ZERO: Seite wird mit Nullen gefüllt
  - __GFP_HIGHMEM: Zugriff auf high memory
  - __GFP_DMA: Zugriff auf DMA-Zone
  - __GFP_HIGH: Zugriff auf emergency memory
- Verknüpfung von Flags durch **bitwise-Oder**:
  - GFP_KERNEL: (__GFP_WAIT |__GFP_IO |__GFP_FS)
    Normale Allocation von Kernel-Speicher; kann warten
  - GFP_USER: Wie GFP_KERNEL, aber für User-Space
  - GFP_ATOMIC: (__GFP_HIGH)
    Höchste Priorität, darf nicht schlafen, Nutzung z.B. von Interrupt-Handlern

---

Alle Flags: `http://lxr.free-electrons.com/source/include/linux/gfp.h`

kmalloc: Retry

Was tue ich, wenn ich dringend Speicher brauche, aber trotzdem NULL bekomme? Gehe ich heulen?

- __GFP_REPEAT: „Versuchs mit mehr Nachdruck“; kann trotzdem schieflaufen
- __GFP_NOFAIL: „Du darfst nicht scheitern!“
- __GFP_NORETRY: „Hat doch alles keinen Sinn, gib einfach auf“

kmalloc I

- Wir erinnern uns, `kmalloc`...
  - gibt **physisch und virtuell** zusammenhängenden Speicher zurück
- Das ist nicht nur ein Vorteil, Bild: Es sind mehr als sechs Pages frei, `kmalloc` gibt aber `NULL` zurück
- architekturabhängiges Limit: komplett portierbarer Code darf nicht mit mehr als 128 KB rechnen

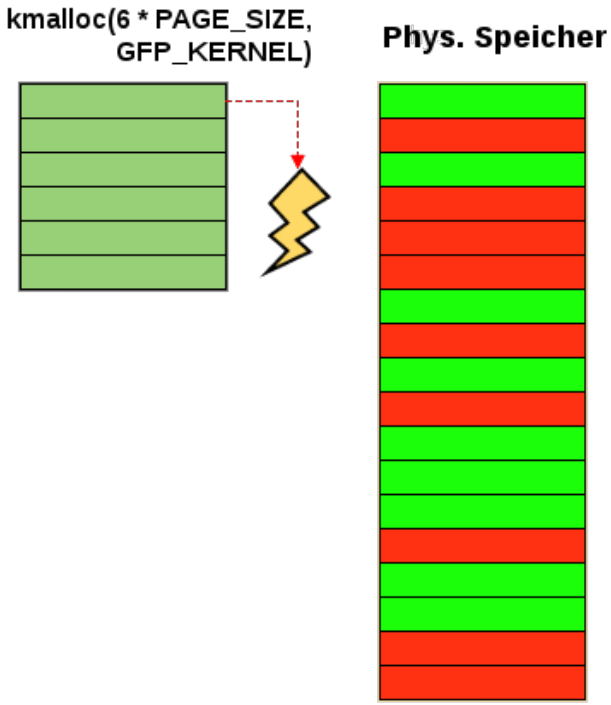

Abbildung: kmalloc: Versuch, 6 Pages zu bekommen

kmalloc II

- je nach Flags
  - **hohe NULL-Quote**
    (⇒ __GFP_DMA, dort zu wenig frei etc.)
  - **Programm wird blockiert**, um auf freie Pages zu warten; kann unerwünscht sein (__GFP_WAIT)
  - **Nebeneffekte** in Low-Memory-Situationen (z.B. __GFP_NOFAIL, __GFP_ATOMIC)
  - **Endlosrekursion**: Kernel-Code im Dateisystem ohne GFP_NOFS: Allocation resultiert in Dateisystem-Operationen, der Code ist selbst beteiligt, allociert Speicher, Dateisystem-Zugriffe, Code ist selbst...
  - **Deadlocks**: Speicher-Bedarf muss durch Swapping gedeckt werden, der Treiber benötigt dafür Speicher

Beispiel: vmalloc

```
1  static int __init vmalloc_test_init(void) {
2    int *ptr = vmalloc(sizeof(int));
3
4    if (ptr == NULL) {
5      return -1;
6    }
7
8    *ptr = 42;
9    // do stuff with ptr...
10   printk(KERN_INFO "ptr is %d", *ptr);
11   vfree(ptr);
12
13   return 0;
14 }
```

vmalloc

- Wir erinnern uns, `vmalloc`...
  - gibt **virtuell** zusammenhängenden Speicher zurück
- Der Kernel kann viele verstreute physische Pages auf einen virtuellen Bereich mappen (Bild)
- Es lassen sich relativ große Speichermengen allocieren (nett)
- Kommt mit hoher Erfolgswahrscheinlichkeit (nett)

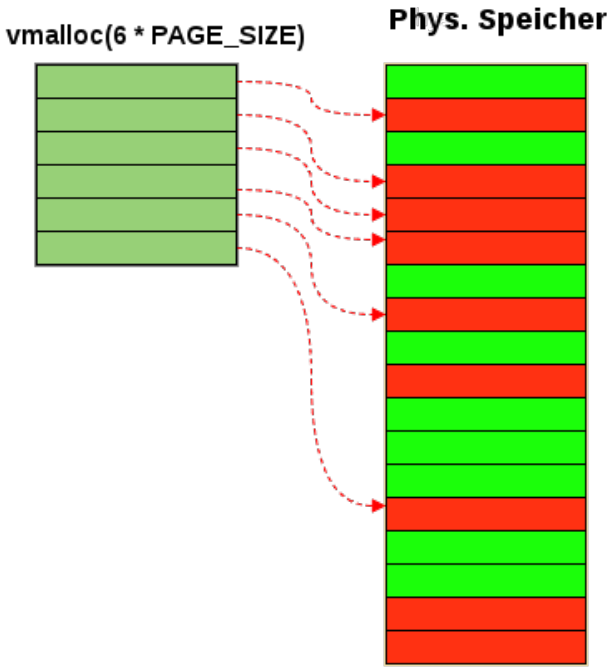

Abbildung: vmalloc: Versuch, 6 Pages zu bekommen

Warum nutze ich dann nicht immer `vmalloc`?

- Code, der `vmalloc` nutzt, ist „likely to get a chilly reception if submitted for inclusion in the kernel“[1]
- Virt. Speicher muss erst durch physischen Speicher abgedeckt werden
  - Neu-Anlegen von *Page Tables*
  - Die Allocation für die Page Tables nutzt intern `kmalloc(..., GFP_KERNEL)` (kann blocken!)
  - Während der Blockade sucht der Kernel freien Speicher, indem er Buffer leert oder User-Prozesse swapped (sehr teuer)
  - Mapping der virt. auf phys. Seiten + Aktualisieren aller beteiligten Instanzen
- `vmalloc` kann nicht in einem atomaren Kontext genutzt werden (da es blockt)
- `vmalloc` ist teuer, vor allem für einzelne Pages

[1] [CRK05, p. 225]

Konsenquenzen: Effiziente Kernel-Programmierung I

Annahme: `data` beinhaltet 100 MB Daten; gegeben:
`void compress(void *src, void *dest)`

```
1  static int __init compressor_init(void) {
2    void *data = gegeben;
3    void *destination = kmalloc(100M, GFP_KERNEL);
4
5    if (destination == NULL) {
6     return -1;
7    }
8
9    compress(data, destination);
10   kfree(destination);
11 }
```

- Ganz schlecht
  - kmalloc muss durch die Menge an physischem Speicher abgedeckt sein; so viel Speicher wird aber nicht verfügbar sein
  - Ausprobiert: `insmod: ERROR: could not insert module kmalloc.ko: Operation not permitted`
  - Alternative: vmalloc
  - Auch sehr unwahrscheinlich

Konsenquenzen: Effiziente Kernel-Programmierung II

```
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 }
```

- Schlecht
  - Mehrmaliger Aufruf von vmalloc (teuer)
  - Der Kernel muss bei jedem compress physischen Speicher mappen (teuer)
  - Wenn ein vmalloc scheitert, scheitert das Modul (doof)

Konsenquenzen: Effiziente Kernel-Programmierung III

```
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 }
```

- Besser
  - **Einmalige Allocation** eines *working memory*
  - Wird **wiederverwendet**
  - Der Kernel muss nur einmal physischen Speicher mappen
  - Wenn das Modul scheitert, dann am Anfang

Welche Fragen sollte ich mir stellen?

- Ist es wichtig, physisch zusammenhängenden Speicher zu bekommen?
- Wie viel Speicher brauche ich?
- Will ich eine Page oder mehrere?
- Brauche ich einen großen Bereich oder mehrere kleine?
- Tun mir die Extra-Kosten von vmalloc (Mapping) weh?
- Welche Auswirkungen hat es, wenn ich keinen Speicher bekomme?
- Welche Art von Speicher will ich?
- etc.

Nochmal zum Beispiel... Weitere Stellschrauben

```
1 void *workmem = kmalloc(PAGE_SIZE, GFP_KERNEL);
2
3 for (each part of data of size PAGE_SIZE) {
4   compress(current part of data, workmem);
5 }
```

oder

```
1 void *workmem = kmalloc(3 * PAGE_SIZE, GFP_KERNEL);
2
3 for (each part of data of size 3 * PAGE_SIZE) {
4   compress(current part of data, workmem);
5 }
```

oder

```
1 void *workmem = vmalloc(10 * PAGE_SIZE);
2
3 for (each part of data of size 10 * PAGE_SIZE) {
4   compress(current part of data, workmem);
5 }
```

Zusammenfassung: Was haben wir gelernt?

- Es gibt verschiedene Arten von Speicherreservierung

  Im Sinne der Effizienz ist nur dynamische Allocation (zur Laufzeit) interessant

- Im User-Space ist Speicherverwaltung einfach

  Wir haben malloc und das Mitdenken (z.B. Heap-Vergrößerung) übernimmt der Kernel

- Im Kernel ist Vorsicht geboten

  Fehler in der Speicherverwaltung haben fatale Folgen (Kernel panic, Datenverlust, Unbenutzbarkeit)

- Es existieren zwei Haupt-Allocatoren im Kernel

  vmalloc (virt. zusammenhängend, große Mengen) und kmalloc (physisch zusammenhängend, bietet Flags)

- Im Kernel muss ich mir mehr Fragen stellen als „wie viel?“

  kmalloc vs. vmalloc, physisch zusammenhängend, wie viele Seiten, zu welchen Kosten usw.

Quellenangaben I

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

[CRK05] Jonathan Corbet, Alessandro Rubini und Greg Kroah-Hartmann. *Linux device drivers*. third. O'Reilly und Associates, Feb. 2005.

[dir] directindustry. URL: http://img.directindustry.com/images_di/kwref/kwref-g/4/6/78064.jpg.

[Fer14] Brice Fernandes. *No title*. Okt. 2014. URL: http://fractallambda.com/2014/10/30/Dynamic-Static-and-Automatic-memory.html.

[IG16] The IEEE und The Open Group. *malloc*. 2016. URL: http://pubs.opengroup.org/onlinepubs/009695399/functions/malloc.html (abgerufen am 16. 12. 2016).

[ker] kernel.org. URL: https://www.kernel.org/theme/images/logos/tux.png.

[mem] memegenerator. URL: https://imgflip.com/memegenerator.

Quellenangaben II

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

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

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

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

[Ott16] Alan Ott. *Virtual Memory and Linux*. Apr. 2016. URL: https://events.linuxfoundation.org/sites/events/files/slides/elc_2016_mem.pdf.

[Pro05] The Linux Information Project. *Kernel*. Mai 2005. URL: http://www.linfo.org/kernel.html.

Quellenangaben III

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

[Zah14] Ivan Zahariev. *Know your Linux memory usage*. Sep. 2014. URL: https://blog.famzah.net/2014/09/22/know-your-linux-memory-usage/.