Speicheranforderungen im Proc-Filesystems analysieren

# Spiegelbilder

Etwas Detailwissen darüber, wie Prozesse Speicher anfordern, hilft beim Debugging unwilliger oder übermäßig RAM-hungriger Prozesse. Obgleich es mehrere Formen zu unterscheiden gilt – alle Informationen spiegelt der Kernel augenblicklich ins Proc-Filesystem ein. Willi Nüßer

**Der Linux-Kernel** lässt über das Proc-Filesystem seiner Außenwelt eine Vielfalt an Informationen über alle Prozess zukommen. So spiegelt die Datei »/proc/*Prozessnummer*/maps« die augenblickliche Gestalt des virtuellen Adressraums des betreffenden Prozesses wider, das so genannte Speicherbild oder die Memory Map [1]. Das Bild erlaubt es Programmierern und Systemadministratoren, die aktuellen Speicherbedürfnisse des Prozesses zu verstehen. Sogar Rückschlüsse auf Fehler, etwa nicht (mehr) zu befriedigende Speicheranforderungen, sind möglich. Mit gängigen Debugging-Werkzeugen ist solchen Problemen nicht so einfach beizukommen.

Um den Inhalt der Maps-Datei verständlich zu machen, beginnt der Artikel mit ein paar Grundlagen zum Kernel. Dann zeigt ein einfaches C-Programm, wie das Speichern eines Prozesses üblicherweise aussieht. Auf diesem Fundament aufbauend werden komplexere Speicheroperationen transparent: Anforderungen von Heap-Speicher, das Einblenden von Dateien in den virtuellen Adressraum und schließlich das Shared Memory.

## Jeder bekommt seinen eigenen Speicher

Linux folgt dem Konzept des Virtual Memory: Jeder Prozess erhält seinen eigenen logischen Adressraum und nur für ihn erzeugen der Compiler und der Linker Adressen. Greift ein Prozesses auf einen Ort in seinem Adressraum zu, zum Beispiel auf eine Variable, setzen Betriebssystem und Prozessor den Aufruf auf deren tatsächliche physikalische Adresse um [1]. Wichtig: Ohne besonderes Zutun sind die Adressräume eines Prozesses nicht von anderen Prozessen aus zugänglich.

Linux teilt den virtuellen Adressraum eines Prozesses in einen für den User zugänglichen Teil und einen dem Kernel vorbehaltenen Bereich. Die Lage der Grenze zwischen beiden variiert unter anderem mit der eingesetzten Hardware-Plattform: Auf einer 32-Bit-Maschine liegt sie meist bei 3 GByte (hexadezimal: 0xC0000000, Makro PAGE_OFFSET im Kernel, [1]). Unterhalb liegt der User-, oberhalb der Kernel-Bereich.

Diese Einteilung pflanzt sich im User-Bereich fort: Der verfügbare Adressraum besteht aus einzelnen Virtual Memory Areas (VMAs). Jede VMA ist ein zusammenhängender Bereich von Adressen, die die gleiche Semantik besitzen. Code- und Daten-Segment eines Prozesses sind Beispiele für VMAs. Eine VMA ist durch folgende Attribute charakterisiert:

- Ihre Start- und Endadresse. Deren Größe ist damit auch bestimmt. In der Praxis variieren die genauen Werte von Start- und Endadresse etwas mit der Kernelversion und der verwendeten Systembibliothek.

Sitzung Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe

```
wn@wnlinux:-> cat /proc/5004/maps
08048000-08049000 r-xp 00000000 03:0a 34167      /data/tmp/maps
08049000-0804a000 rw-p 00000000 03:0a 34167      /data/tmp/maps
40000000-40014000 r-xp 00000000 03:09 85209      /lib/ld-2.3.2.so
40014000-40015000 rw-p 00014000 03:09 85209      /lib/ld-2.3.2.so
40015000-40016000 rw-p 00000000 00:00 0
40028000-40157000 r-xp 00000000 03:09 9443       /lib/libc.so.6
40157000-4015b000 rw-p 0012f000 03:09 9443       /lib/libc.so.6
4015b000-4015e000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
wn@wnlinux:->
```

Neu | Befehlsfenster

**Abbildung 1:** Darstellung des Speicherbilds eines Prozesses in »/proc/*Prozessnummer*/maps«.

- Ihre Rechte. Eine VMA, die Code enthält, besitzt die Rechte »r-xp«: Sie ist lesbar, nicht beschreibbar, aber ausführbar und nur für diesen Prozess sichtbar (privat).
- Informationen über den so genannten Backing Store, den der Linux-Kernel auch unter dem Begriff »vm_file« führt. Sie spezifizieren, ob und wo die VMA ein Gegenstück auf dem Dateisystem besitzt. Dieses Pendant dient als Auffangbecken beim Auslagern der VMA aus dem Speicher. Bei den Code- und Daten-VMAs ist der Backing Store die Programm-Binary selbst.

## Speicherbild visualisieren

Die Datei »/proc/*Prozessnummer*/maps« listet die aktuellen VMAs. Über sie macht der Kernel einen Großteil der Informationen, die er über den virtuellen Adressraum des Prozesses hat, anderen Programmen und Usern zugänglich. Das bewusst einfach gehaltene Testprogramm in Listing 1 nutzt dies aus.

Der Prozess gibt nur seine eigene Prozess-ID (PID) aus und wartet dann 30 Sekunden, um dem Benutzer Gelegenheit zu geben, per »cat /proc/*Prozessnummer*/maps« das Speicherbild in einem anderen Terminal auszugeben. Abbildung 1 zeigt das Ergebnis für einen Prozess mit der PID 5004.

Die ersten beiden Zeilen der Liste spiegeln das Code- beziehungsweise das Daten-Segment wider. Die erste Hexadezimalzahl in jeder Zeile (beispielsweise »08048000«) gibt den Beginn der VMA an, die nächste Zahl das Ende. Die folgende Zeichenkette enthält die erwähnten Attribute der VMA. Hier ist der Unterschied zwischen dem Code-Segment in der ersten und dem Daten-Segment in der zweiten Zeile zu sehen.

Die nächste Hex-Zahl gibt den Offset in der zugrunde liegenden Datei (Backing Store) an. Die Angabe »03:0a« liefert die Major- und die Minor-Nummer des Geräts, auf dem der Backing Store liegt. Auf dem System des Autors liefert »ls -la« im »/dev«-Verzeichnis unter anderem:

```
brw-rw----    1 root     disk       3,↩
  10 Mar 14  2003 /dev/hda10
```

Damit liegt der Backing Store, hier die Programmdatei, auf der Partition »/dev/hda10«. Die Inode-Nummer und der Name der Datei folgen.

## Block Started by Symbol

Die Zeilen 3 bis 5 beziehungsweise 6 bis 8 der Liste tauchen in dieser Form bei jedem Linux-Programm auf. Die erste Gruppe gehört zum dynamischen Linker (»ld.so«), die zweite zur Glibc. Beiden Bibliotheken ist neben ihrem jeweiligen Code- und Daten-Segment noch eine dritte VMA zuzuordnen: Zeile 5 dem Linker, Zeile 8 der Glibc.

Beide VMAs bezeichnet die Unix-Terminologie als BSS-Bereich (Block Started by Symbol). Hier liegen – wenn vorhanden – globale Daten, die zum Programmstart zwar allokiert, aber noch nicht initialisiert sind. Diese VMAs besitzen keinen Speicherort im Filesystem, sie benutzen stattdessen den Swap-Bereich.

Die letzte Zeile stellt die VMA des Stacks dar. Bei Linux auf Intel-Plattformen beginnt der Stack an der Grenze zwischen User- und Kernel-Bereich bei 3 GByte und wächst nach unten. Auffallend sind hier weniger die Größe (0xC0000000 bis 0xBFFFE000 entsprechen 8 KByte) als die Attribute: Der Stack ist ausführbar. Das ist eine Voraussetzung für Stack-basierte Buffer-Overflow-Attacken! Die Abbildung 2 fasst diese und einige der später erwähnten Speicherbereiche in einer Grafik zusammen.

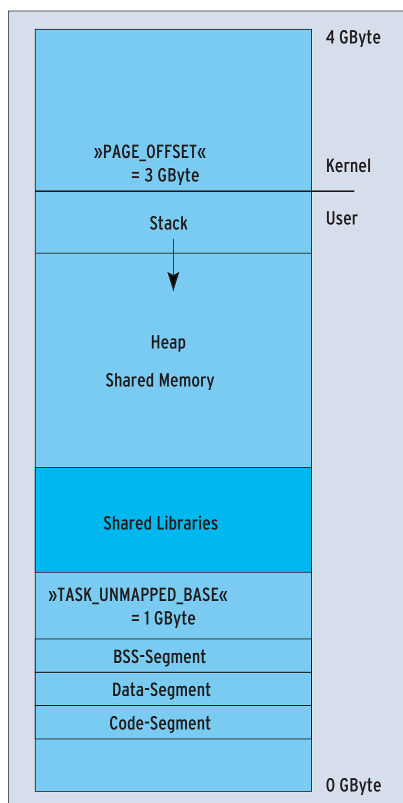


**Abbildung 2: Logische Gestalt des virtuellen Adressraums eines Prozesses unter Linux.**

## Neuer Kernel, neue Virtual Memory Area

Im 2.6er Kernel kommt eine weitere VMA im – dem Kernel vorbehaltenen – oberen Teil des virtuellen Adressraums hinzu: Zwischen 0xFFFFE000 und 0xFFFFF000 liegt eine 4-KByte-Page zum Abwickeln der so genannten Vsyscalls (Virtual Syscalls). Über diese Kernelpage darf ein Prozess auf manche Kernel-interne Daten direkt zugreifen, beispielsweise den CPU-Counter, was Zeit spart

**Listing 1: Grundprogramm**

```
01 #include <sys/types.h>
02 #include <sys/shm.h>
03 #include <sys/ipc.h>
04 #include <time.h>
05 #include <unistd.h>
06 #include <stdio.h>
07 #include <stdlib.h>
08
09 int main() {
10
11   /* return PID for checking */
12   printf("My Pid is: %d\n", getpid());
13   fflush(stdout);
14
15   sleep(30);
16
17   return(0);
18 }
```

**Listing 2: Globale und automatische Variablen**

```
01 ...
02 #define GLEN 100000
03 #define SLEN 12000
04
05 char gfeld [GLEN];
06
07 int main() {
08
09   char afeld[SLEN];
10
11   /* return PID for checking */
12   printf("My Pid is: %d\n", getpid());
13   fflush(stdout);
14
15   sleep(30);
16
17   return(0);
18 }
```

gegenüber einem gleichwertigen normalen Systemaufruf per Interrupt. [2]

## Mehr Speicher!

Aufbauend auf dem ersten Miniprogramm (Listing 1) allokiert das Programm aus Listing 2 Speicher auf gleich zwei Arten:

- Zeile 5 legt ein globales – nicht-initialisiertes – Character-Feld der Größe 100000 an. Dieser Speicherplatz sollte, den obigen Überlegungen folgend, im BSS-Segment auftauchen und so zu einer neuen VMA führen (analog kämen statische Variablen in Frage).
- Zeile 9 erzeugt ein Character-Feld der Größe 12000. Die Variable »afeld[]« gehört damit zu den lokalen oder Automatic-Variablen und landet auf dem Stack.

Listing 3 zeigt das zugehörige Speicherbild. Im Stack-Bereich sind drei zusätzliche Pages allokiert, um die 12000 Bytes unterzubringen (0xBFFFB000 statt 0xBFFFE000). Zeile 3 symbolisiert die neue VMA für die nicht initialisierten globalen Daten.

Hier wird allerdings nicht der gesamte Bereich neu angefordert, sondern etwas weniger: 98304 Bytes. Die Erklärung für diesen Fehlbetrag findet sich, wenn man Strace [3] auf die Abläufe der beiden bisherigen Programmversionen ansetzt: In beiden Fällen passiert die Allokierung des Speichers für die Daten über einmaliges Ausführen des Systemaufrufs »brk()«. Der Aufruf verschiebt die Spitze des für Daten allokierten Speicherbereichs nach oben.

**Listing 3: Speicherbild zu Listing 2**

```
01 08048000-08049000 r-xp 00000000 03:0a 34167      /data/tmp/maps
02 08049000-0804a000 rw-p 00000000 03:0a 34167      /data/tmp/maps
03 0804a000-08062000 rwxp 00000000 00:00 0
04 40000000-40014000 r-xp 00000000 03:09 85209      /lib/ld-2.3.2.so
05 40014000-40015000 rw-p 00014000 03:09 85209      /lib/ld-2.3.2.so
06 40015000-40016000 rw-p 00000000 00:00 0
07 40028000-40157000 r-xp 00000000 03:09 9443       /lib/libc.so.6
08 40157000-4015b000 rw-p 0012f000 03:09 9443       /lib/libc.so.6
09 4015b000-4015e000 rw-p 00000000 00:00 0
10 bfffb000-c0000000 rwxp ffffc000 00:00 0
```

**Listing 4: Malloc verwenden**

```
01 ...
02 #define MLEN 1
03
04 int main() {
05   void *vp;
06   vp = malloc(MLEN);
07
08   /* return PID for checking */
09   printf("My Pid is: %d\n", getpid());
10   fflush(stdout);
11
12   sleep(30);
13   return(0);
14 }
```

**Listing 5: Speicherbild zu Listing 4**

```
01 08048000-08049000 r-xp 00000000 03:0a 34167      /data/tmp/list4
02 08049000-0804a000 rw-p 00000000 03:0a 34167      /data/tmp/list4
03 0804a000-0804b000 rwxp 00000000 00:00 0
04 40000000-40014000 r-xp 00000000 03:09 85209      /lib/ld-2.3.2.so
05 40014000-40015000 rw-p 00014000 03:09 85209      /lib/ld-2.3.2.so
06 40015000-40016000 rw-p 00000000 00:00 0
07 40028000-40157000 r-xp 00000000 03:09 9443       /lib/libc.so.6
08 40157000-4015b000 rw-p 0012f000 03:09 9443       /lib/libc.so.6
09 4015b000-4015e000 rw-p 00000000 00:00 0
10 bfffe000-c0000000 rwxp fffff000 00:00 0
```

**Listing 6: Speicherbild bei MLEN=300000**

```
01 08048000-08049000 r-xp 00000000 03:0a 34167      /data/tmp/list4_2
02 08049000-0804a000 rw-p 00000000 03:0a 34167      /data/tmp/list4_2
03 0804a000-0804b000 rwxp 00000000 00:00 0
04 40000000-40014000 r-xp 00000000 03:09 85209      /lib/ld-2.3.2.so
05 40014000-40015000 rw-p 00014000 03:09 85209      /lib/ld-2.3.2.so
06 40015000-40016000 rw-p 00000000 00:00 0
07 40028000-40157000 r-xp 00000000 03:09 9443       /lib/libc.so.6
08 40157000-4015b000 rw-p 0012f000 03:09 9443       /lib/libc.so.6
09 4015b000-401a8000 rw-p 00000000 00:00 0
10 bfffe000-c0000000 rwxp fffff000 00:00 0
```

## Bytes sparen

Dabei – und das ist wesentlich – gibt es keine Unterscheidung zwischen den normalen globalen Daten des Daten-Segments und den Daten des BSS-Segments. Eine kluge Allokierungspolitik hilft im vorliegenden Fall einige Bytes sparen, da ein Teil des Daten-Segments für BSS nutzbar wird. Das bedeutet, dass ein Wert von GLEN = 1 normalerweise nicht zur Allokierung einer neuen Page im BSS-Bereich führt.

Doch was passiert, wenn GLEN so groß wird, dass es in den Bereich ragt, in dem die Bibliotheken beginnen? Diesen Startpunkt legt der Linux-Kernel mit der Konstante »TASK_UNMAPPED_BASE« auf »PAGE_OFFSET / 3« fest – im vorliegenden Fall bei 1 GByte. Das Anfordern reichlicher 930 MByte innerhalb eines globalen Feldes führt dann auch zu einem Speicherfehler, dem berühmten SIGSEGV.

## Arbeiten im Heap: Die Bibliotheksfunktion Malloc

Kein ernst zu nehmendes Programm kommt mit den beschriebenen Formen der Speichernutzung aus. Denn oft wird die benötigte Speichermenge erst zur Laufzeit des Programms klar. Um dynamische Anforderungen erfüllen zu können, stellt das Betriebssystem einen weiteren Bereich zur Verfügung, den so genannten Heap. Aus ihm kann sich ein Prozess zum Beispiel über die Malloc-Familie mit Speicher versorgen.

Im Gegensatz zum System-Aufruf »brk()« ist »malloc()« eine Bibliotheksfunktion. Je nach Parameter variiert die Bibliothek die Arbeitsweise von Malloc. Das leicht erweiterte Testprogramm aus Listing 4 hilft dynamische Speicheranforderungen verstehen. Führt man es aus, entsteht das Speicherbild aus Listing 5. Auffällig ist, dass Linux bereits für 1 Byte eine neue VMA von der Größe einer Page angelegt (0x0804A000 bis 0x0804B000). Es zieht also nicht wie eben das ohnehin vorhandene Daten-Segment bei der Adresse 0x08049000 heran.

Trotzdem zeigt ein Tracing des Programms, dass weiterhin nur »brk()« – wenn auch mit veränderten Argumenten – aufgerufen wird. Das ändert sich so lange nicht, bis der angeforderte Speicher eine Grenze nicht überschreitet, die bei der verwendeten Glibc-Version 2.3.2 auf 128 KByte voreingestellt ist. Das Limit lässt sich ändern, indem man der Bibliotheksfunktion »mallopt()« über den Parameter »M_MMAP_THRESHOLD« einen neuen Wert gibt [4].

## Wenn's zu viel wird

Übersteigt eine Anforderung den eingestellten Wert, schaltet die Glibc auf einen zweiten – mitunter langsameren – Modus um. Um das zu demonstrieren,

reicht es, im Listing 4 das Makro »MLEN« auf 300000 zu setzen. Dabei entsteht das Speicherbild von Listing 6. Die Page aus Zeile 3 ist aus dem ersten Malloc-Beispiel bekannt und wird wie dort über »brk()« allokiert. Der Löwenanteil des angeforderten Speichers liegt nun aber zwischen 0x4015B000 und 0x401A8000. Das ist der BSS-Bereich der Glibc selbst. Eine einfache Rechnung zeigt, dass 77 Pages angefordert wurden: Abzüglich der drei, die die Glibc ohnehin schon besaß, sind das genau die 74 neuen Pages (= 303104 Bytes), die mindestens notwendig werden, um 300000 Bytes in 4-KByte-Pages unterzubringen.

## Bei 2 GByte ist auf 32-Bit-Systemen Schluss

Genau an dieser Stelle beginnt das Hauptproblem von speicherintensiven Anwendungen auf 32-Bit-Plattformen: Sobald ein Programm seine Speicheranforderung so weit hinaufsetzt, dass es in den Stack-Bereich hineinragt, löst das den Speicherfehler SIGSEGV aus und der Prozess terminiert. Per »malloc()« allokierbar sind somit nur rund 2 GByte – zu wenig für größere Datenbanken, Applikationsserver und Multimedia-Anwendungen. Diese Beschränkung gilt leider nicht nur für Malloc, sondern auch für alle anderen Formen der Speicheranforderung (siehe unten).

Noch offen ist, wie die Zuweisung von Speicher von 0x4015B000 bis 0x401A8000 passiert. Hier hilft es, das Programm abermals mit Strace zu untersuchen. Als entscheidende Zeile kommt hinzu:

```
old_mmap(NULL, 303104,
PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS,-1, 0)
```

Als Ergebnis liefert der Systemruf 0x4015e000. Die Glibc verwendet hier eine Form des so genannten Memory Mapping (Mmap). Die zurückgelieferte Adresse 0x4015e000 entspricht der Endadresse, die die Glibc bislang für sich gesetzt hat. Der angeforderte Bereich, der im Übrigen die oben berechnete Größe hat, liegt jetzt genau hinter dem bisherigen Glibc-Bereich.

## Dateien einblenden per Mmap-Systemruf

Eine Alternative zum Anlegen einer neuer VMA ist es, eine Datei (oder ein anderes Objekt) in den virtuellen Adressraum einzublenden. Dies Memory Mapping wird gerne eingesetzt, da ein Prozess statt mit den normalen Read- und Write-Operationen über Pointer direkt auf die Datei zugreifen kann. Das Betriebssystem stellt dann die Daten bereit; bei Abwesenheit sorgen Page Faults dafür, dass der Kernel die angefragten Teile in den Speicher bringt. ▶

**Listing 7: Mmap**

```
01 ...
02 #define FLEN 7000
03
04 int main() {
05   int ffd;
06   void *vp1;
07
08   ffd = open("/tmp/bla", O_RDWR);
09   vp1 = mmap(0, FLEN, PROT_READ | PROT_WRITE, MAP_PRIVATE, ffd, 0);
10
11   /* return PID for checking */
12   printf("My Pid is: %d\n", getpid());
13   fflush(stdout);
14
15   sleep(30);
16
17   munmap(vp1, FLEN);
18   return(0);
19 }
```

**Listing 8: Speicherbild zu Listing 7**

```
01 08048000-08049000 r-xp 00000000 03:0a 47891    /data/tmp/list7
02 08049000-0804a000 rw-p 00000000 03:0a 47891    /data/tmp/list7
03 40000000-40014000 r-xp 00000000 03:09 85209    /lib/ld-2.3.2.so
04 40014000-40015000 rw-p 00014000 03:09 85209    /lib/ld-2.3.2.so
05 40015000-40017000 rw-p 00000000 03:09 179700   /tmp/bla
06 40017000-40018000 rw-p 00000000 00:00 0
07 40028000-40157000 r-xp 00000000 03:09 9443     /lib/libc.so.6
08 40157000-4015b000 rw-p 0012f000 03:09 9443     /lib/libc.so.6
09 4015b000-4015e000 rw-p 00000000 00:00 0
10 bfffe000-c0000000 rwxp fffff000 00:00 0
```

- Anzeige -

Linux schreibt die gemappten Daten auch selbsttätig auf die Datei zurück, synchronisiert also Hauptspeicher und Datei-Inhalt miteinander. Die Grundlage dieses Verfahren bildet »mmap()«. Auf einer Intel-Maschine implementiert Linux Kernel-intern Mmap durch die Funktion »old_mmap()«, die eben schon kurz Thema war. Beim Aufrufen erwartet »mmap()« einige Argumente:

- Die Startadresse des einzublendenden Bereichs in den virtuellen Adressraum. Ist der Wert wie oben auf »NULL« gesetzt, bestimmt das Betriebssystem die Adresse selbst.
- Die Größe des Bereichs. Bei einer Datei als Ausgangsobjekt darf der Wert auch kleiner als die Datengröße sein.
- Die Zugriffsrechte der erzeugten VMA, zum Beispiel »READ« und »WRITE«.
- Die Attribute der VMA, vor allem, ob sie nur diesem Prozess (privat) oder für mehrere Prozesse gemeinsam (shared) zugänglich wird. Ein anderes Attribut bestimmt, ob das eingeblendete Objekt eine reale Datei ist. Andernfalls verbleiben die in diese VMA geschrieben Daten nur temporär im Speicher, genauer im Page-Cache oder Swapspace. Dann liegt ein unbenannter (anonymer) Speicherbereich vor.
- Den Filedeskriptor der einzublendenden Datei, die man zuvor mit »open()« erzeugt hat. Anonymer Speicher bekommt »-1« als Argument übergeben.
- Bei Dateien ist optional ein Offset bestimmbar, ab dem Mmap die geforderte Größe einblenden soll (siehe [5] bis [7]).

**Listing 9: Anonymes Shared Mapping**

```
01 08048000-08049000 r-xp 00000000 03:0a 47891      /data/tmp/mmapt
02 08049000-0804a000 rw-p 00000000 03:0a 47891      /data/tmp/mmapt
03 40000000-40014000 r-xp 00000000 03:09 85209      /lib/ld-2.3.2.so
04 40014000-40015000 rw-p 00014000 03:09 85209      /lib/ld-2.3.2.so
05 40015000-40017000 rw-s 00000000 00:05 14648      /dev/zero
   (deleted)
06 40017000-40018000 rw-p 00000000 00:00 0
07 40028000-40157000 r-xp 00000000 03:09 9443       /lib/libc.so.6
08 40157000-4015b000 rw-p 0012f000 03:09 9443       /lib/libc.so.6
09 4015b000-4015e000 rw-p 00000000 00:00 0
10 bfffe000-c0000000 rwxp fffff000 00:00 0
```

**Listing 10: SysV-SHM**

```
01 ...
02 #define SLEN 9000
03 #define KEY 4242
04 int main() {
05   int hdl;
06   void *vp1;
07   key_t key;
08   struct shmid_ds shminfo;
09
10   key = KEY;          /* Huh */
11   printf("Der Key: %x\n", key);
12   hdl = shmget(key, SLEN, 0660 | IPC_CREAT);
13   printf("Die SHM ID: %d\n", hdl);
14
15   vp1 = shmat(hdl, NULL, 0);
16
17   /* return PID for checking */
18   printf("My Pid is: %d\n", getpid());
19   fflush(stdout);
20
21   /* write into SHM Segment */
22   strncpy((char*)vp1, "Hallo Welt", 11);
23
24   sleep(30);
25
26   shmdt(vp1);
27   shmctl(hdl, IPC_RMID, &shminfo);
28   return(0);
29 }
```

Listing 7 demonstriert die recht komplexe Semantik. Zu Beginn öffnet es eine Datei »/tmp/bla« zum Lesen und Schreiben, die es dann in den virtuellen Adressraum des Prozesses einblendet. Um die 7000 Bytes abzudecken, bedarf es zweier Pages.

Zum Ende des Programms hebt »munmap()« diese Abbildung wieder auf. Das wäre hier zwar unnötig, da es bei Prozessende automatisch geschieht, würde in Produktivcode aber das Verschmutzen des Adressraums wirksam verhindern.

Das zugehörige Speicherbild ist in Listing 8 ersichtlich. Es zeigt die Datei und deren Rechte inklusive dem Privat-Bit (»rw-p«). Offenbar werden die Code- und Datenbereiche selbst über den gleichen »mmap«-Mechanismus eingeblendet.

Der einzige Unterschied zwischen Code- und Datenbereich und dem eigenhändig eingeblendeten File ist die Lage: Bei einem expliziten »mmap()« ohne Angabe einer Zieladresse beginnt der Kernel mit der Suche nach einer passenden Stelle erst bei »TASK_UNMAPPED_BASE«.

## Ohne Datei: Impliziertes Mapping

Die implizite Form des Mapping beim Malloc-Aufruf unterscheidet sich nur durch die Angabe des zusätzlichen Attributs »MAP_ANONYMOUS«. Das folgende Beispiel setzt dieses Flag explizit. Zudem soll auch der Bereich von mehreren Prozessen gemeinsam nutzbar werden. Dazu ersetzt

```
vp1 = mmap(NULL, FLEN, PROT_READ | ➘
PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS,-1,➘
 0);
```

das bisherige »mmap()« in Zeile 9 (Listing 7). Der »open()«-Aufruf in Zeile 8 entfällt mangels Bezug zu einer Datei. Im Speicherbild von Listing 9 taucht dann auch statt des Dateinamens die Kennung »/dev/zero (deleted)« auf. Der Eintrag auf »/dev/zero« ist durch die Kombination von »MAP_SHARED« und »MAP_ANONYMOUS« entstanden und der Zusatz »(deleted)« weist darauf hin, dass zu diesem File kein echter Directory-Eintrag existiert. Die Kombination des anonymen mit dem Shared Mapping ist erst seit Kernel 2.4 möglich.

## Speicher gemeinsam nutzen: Shared Memory

Das Beispiel von eben eignet sich als Startrampe für die Erklärung einer anderen Methode, nach der Prozesse untereinander Daten über Speicherbereiche austauschen können. Dieses Shared Memory (SHM) genannte Form ist im Allgemeinen die schnellste Art der Interprozess-Kommunikation [8]. Unix-ähnliche Betriebssysteme haben drei Möglichkeiten, ihren Prozessen SHM zu ermöglichen. Linux beherrscht sie alle:

- Über ein Shared Mmap, wie eben beschrieben. Die beteiligten Prozesse blenden Speicherbereiche ein, die auf eine einzige Datei verweisen.
- Schneller und flexibler ist das Shared Memory nach Unix System V.

```
wn@wnlinux-> ipcs -m
key        shmid     owner     perms     bytes     nattch    status
0x00001092 1343508   wn        660       9000      1
bash$ █
```

Abbildung 3: Nach »shmget()« taucht das Segment im System auf, wie ein »ipcs -m« beweist.

- Seit Kernel 2.4 unterstützt Linux den gemeinsamen Speicher nach dem Posix-Standard.

Der gravierendste Nachteil von Shared Mmap ist die funktional bedingt geringe Geschwindigkeit, mit der die Abgleich-Operationen zwischen dem dahinter liegenden File und dem Speicher ablaufen. Nun läge es nahe, auf impliziertes Mapping (siehe oben) auszuweichen. Das scheitert leider am fehlenden Zusammenhang mit einem File.

Der eingeblendete Speicherbereich ist dadurch für gleichrangige andere Prozesse nicht identifizierbar – Stichwort: anonymes Mapping. Die »MAP_SHARED | MAP_ANONYMOUS«-Form des prozessübergreifenden Speichers ist nur möglich, wenn ein Prozess das Speicherbild des anderen kennt, eine Situation, wie sie nur in Elter-Kind-Beziehungen nach einem »fork()« auftritt [5], [6].

## Geteilter Speicher nach Art des System V

Weniger eingeschränkt als Shared Mmap ist eine Form des gemeinsamen Speichers, die aus dem Unix-System V entstand. SysV-SHM ist der wohl am weitesten verbreitete Shared-Memory-Typ. Im Kern erzeugt man als gemeinsam verwendbaren Speicherbereich ein so genanntes Shared-Memory-Segment und blendet es in den virtuellen Adressraum ein. Im Gegensatz zum anonymen Shared Mmap (siehe oben) ist aber das Segment über eine Kennung, einen Key, eindeutig identifizierbar.

Das Programm aus Listing 10 zeigt eine verständliche Demo-Anwendung für geteilten Speicher nach SysV. Im ersten Schritt beschafft sie sich einen eindeutigen Schlüssel, über den alle Prozesse das spätere SHM-Segment identifizieren können. Dafür eignet sich beispielsweise die Funktion »ftok()« [9]. Listing 10 macht es sich aber ganz einfach und setzt den Key nach Gutdünken – für ein echtes Programm wäre das fieser Stil.

Jeder Prozess, der auf demselben Segment arbeiten will, muss diesen Key kennen. Der Aufruf »shmget(key, SLEN, 0660 | IPC_CREAT)« aus Zeile 12 erzeugt nun ein »SLEN«-großes SHM-Segment zum Lesen und Schreiben (wegen »0660«). Gab es das Segment bereits, verhindert »IPC_CREAT« den Konflikt [10].

Schon an dieser Stelle im Programmablauf taucht das Segment im System auf, wie »ipcs -m« in Abbildung 3 beweist. »ipcs« ist unter Linux das Tool der Wahl für das Anzeigen von Segmenten. Erst im weiteren Verlauf des Programms blendet der Aufruf »shmat()« den Adressraum des Prozesses in den virtuellen Speicher ein. Das einzublendende Segment wird über ein Handle identifizierbar, das »shmget()« zurückgibt. Es folgen die Angabe der gewünschten Zieladresse für die zugehörige VMA und optionale Flags. Das Ergebnis des Anhängens zeigt das Listing 11.

Die sechste Zeile ist neu und weist drei Besonderheiten auf:

- Die Inode-Nummer, die in der vorletzten Spalte steht, entspricht der SHMID von »ipcs«.
- Der Dateiname ist eine Kombination aus der einleuchtenden Bezeichnung »SYSV« und dem Key.
- Da das Segment nicht zu einem echten Directory gehört, taucht auch hier eine »(deleted)«-Marke auf.

Vor Programmende in Listing 10 entfernt »shmdt()« aus Zeile 26 zunächst das SHM-Segment aus dem Adressraum und »shmctl(hdl, IPC_RMID, &shminfo)« nimmt es sogar gänzlich aus dem System. Linux realisiert intern »shmat()« mit dem gleichen Code wie »mmap()«.

Das im Listing 11 sichtbare Pseudo-File »SYSV00001092« liegt in einem speziellen Filesystem des Linux-Kernels, dem Tmp-Filesystem.

## Das flotte Temp-Filesystem und Posix-SHM

Das Tmp-FS feierte im Kernel 2.4 sein Debüt. Es lässt sich wie jedes andere in den Verzeichnisbaum einbauen, lebt aber komplett im Hauptspeicher, das heißt im Pagecache und Swapspace. Das macht das Tmp-FS zum Kandidaten für schnelle Schreib- und Leseoperationen und zudem zur effiziente Basis für gemeinsamen Speicher.

Die eine Variante, um geteilten Speicher im Tmp-FS zu erzeugen, wäre ein

```
mmap(...,MAP_SHARED,...)
```

**Listing 11: Speicherbild zu Listing 10**

```
01 08048000-08049000 r-xp 00000000 03:0a 46930     /data/tmp/list10
02 08049000-0804a000 rw-p 00000000 03:0a 46930     /data/tmp/list10
03 40000000-40014000 r-xp 00000000 03:09 85209     /lib/ld-2.3.2.so
04 40014000-40015000 rw-p 00014000 03:09 85209     /lib/ld-2.3.2.so
05 40015000-40016000 rw-p 00000000 00:00 0
06 40016000-40019000 rw-s 00000000 00:05 1343508   /SYSV00001092 (deleted)
07 40028000-40157000 r-xp 00000000 03:09 9443      /lib/libc.so.6
08 40157000-4015b000 rw-p 0012f000 03:09 9443      /lib/libc.so.6
09 4015b000-4015e000 rw-p 00000000 00:00 0
10 bfffe000-c0000000 rwxp fffff000 00:00 0
```

**Listing 12: Posix-SHM**

```
01 ...
02 #define GLEN 4000
03 #define MLEN 2000
04 /* Note: prefix /dev/shm will be added automatically */
05 #define SHMFILE "meines"
06
07 int main() {
08
09   int hdl;
10   char *vp1;
11   hdl = shm_open(SHMFILE,
12                O_RDWR | O_CREAT, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
13   ftruncate(hdl, GLEN);
14    vp1 = mmap(0, MLEN, PROT_READ | PROT_WRITE, MAP_SHARED, hdl, 0);
15
16   /* return PID for checking */
17   printf("My Pid is: %d\n", getpid());
18   fflush(stdout);
19
20   sleep(30);
21   munmap(vp1, MLEN);
22   shm_unlink(SHMFILE);
23   return(0);
24 }
```

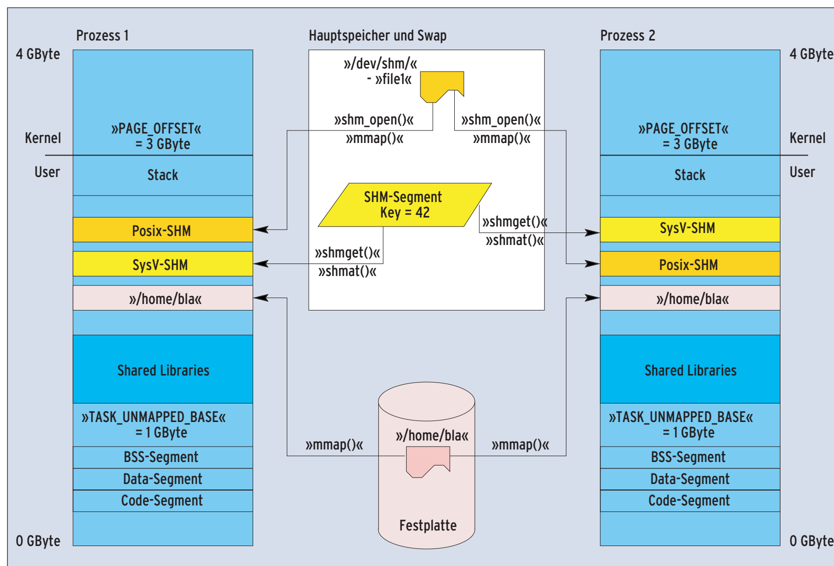


**Abbildung 4: Die drei Formen von Shared Memory ab Linux 2.4.**

auf ein File im Tmp-FS. Eine zweite Spielart ist zwar technisch äquivalent, folgt aber dem Posix-Standard. Dieser Ansatz stand lange im Schatten von SysV-SHM, bietet aber gerade unter Linux vielseitige Möglichkeiten.

Das Posix-SHM-Programm in Listing 12 öffnet zunächst ein Shared Memory Object. Bei Linux ist das eine Datei im Tmp-FS-Dateisystem. Der zugehörige Befehl »shm_open()« [11] folgt der Syntax des »open()«-Aufrufs. Er verwendet stets implizit das Tmp-FS, das unter »/dev/shm« gemountet sein sollte. »shm_open()« legt eine Datei der Länge null in diesem Filesystem an. Das »ftruncate()« weist ihr dann die gewünschte Länge zu [11].

## Der Rest ist bekannt

Das Ein- beziehungsweise Ausblenden in den Adressraum durch »mmap()« und »munmap« ist bekannt. »shm_unlink()« übernimmt die Rolle von »unlink()« [11] und löscht zu guter Letzt die Datei.

**Listing 13: »/dev/shm« zu Listing 12**

```
01 ls -lai /dev/shm/
02 total 93
03     300 drwxrwxrwt    2 root     root          60 Mar 29 16:22 .
04     636 drwxr-xr-x   29 root     root       94888 Mar 29 08:57 ..
05   24425 -rw-r-----    1 wn       users       4000 Mar 29 16:22 meines
```

**Listing 14: Speicherbild zu Listing 12**

```
01 08048000-08049000 r-xp 00000000 03:0a 32347     /data/tmp/list12
02 08049000-0804a000 rw-p 00000000 03:0a 32347     /data/tmp/list12
03 40000000-40014000 r-xp 00000000 03:09 85209     /lib/ld-2.3.2.so
04 40014000-40015000 rw-p 00014000 03:09 85209     /lib/ld-2.3.2.so
05 40015000-40016000 rw-s 00000000 00:09 24425     /dev/shm/meines
06 40016000-40017000 rw-p 00000000 00:00 0
07 40028000-4002e000 r-xp 00000000 03:09 9461      /lib/librt.so.1
08 4002e000-4002f000 rw-p 00005000 03:09 9461      /lib/librt.so.1
09 4002f000-4003a000 rw-p 00000000 00:00 0
10 4003a000-40169000 r-xp 00000000 03:09 9443      /lib/libc.so.6
11 40169000-4016d000 rw-p 0012f000 03:09 9443      /lib/libc.so.6
12 4016d000-40171000 rw-p 00000000 00:00 0
13 40171000-4017e000 r-xp 00000000 03:09 9459      /lib/libpthread.so.0
14 4017e000-4017f000 rw-p 0000d000 03:09 9459      /lib/libpthread.so.0
15 4017f000-401c1000 rw-p 00000000 00:00 0
16 bfffe000-c0000000 rwxp fffff000 00:00 0
```

Sobald das Programm läuft, entsteht das File mit der gewünschten Größe im Tmp-FS, wie Listing 13 beweist.

Das Speicherbild von Listing 14 zeigt diese Datei an der erwarteten Stelle. Die Inode-Nummer stimmt ebenfalls mit der Angabe aus dem Verzeichnis-Listing überein. Die drei neuen Libpthread-Zeilen vor dem Stack-Bereich sind nötig, da wegen »shm_open()« zusätzlich noch mit der Realtime-Bibliothek Librt gelinkt werden muss. Abbildung 4 zeigt zusammenfassend die drei Formen gemeinsamen Speichers.

## Das Handwerkszeug steht für jedermann bereit

Der Artikel hat alle Möglichkeiten, unter Linux Speicher anzufordern, beschrieben. Das Maps-File liefert ein verlässliches Abbild des virtuellen Adressraums eines Prozesses. Mit diesem Wissen und entsprechenden Mitteln, sollten Programmierer und Admins weniger Probleme dabei haben, Speicheranforderungen zu analysieren und mögliche Speicherprobleme – auch bei angeforderten Bibliotheken – zu erkennen. *(jk)* ■

### Infos

[1] Stefan Klett, „So verwaltet der Linux-Kernel den Speicher“: Linux-Magazin 09/03, S. 91
[2] A. Arcangeli über Vsyscalls: [http://www.ukuug.org/events/linux2001/papers/html/AArcangeli-vsyscalls.html]
[3] Uwe Schneider, „Strace – Programmdiagnose für Entwickler und Administratoren“: Linux-Magazin 09/02, S. 96
[4] Mallopt: »info mallopt«
[5] W.R. Stevens, „Advanced Programming in the Unix Environment“: Addison-Wesley, 1992
[6] W.R. Stevens, „Unix Network Programming Volume 2, Interprocess Communication“: Prentice Hall, 1999
[7] Mmap: »man mmap«
[8] D. Henrici, „Standard Template Library für Objekte im Shared Memory verwenden“: Linux-Magazin 12/03, S. 102
[9] Ftok: »man ftok«
[10] Shmget und Shmat: »man shmget« und »man shmat«
[11] Shm_open, Ftruncate und Shm_unlink: »man shm_open«, »man ftruncate« und »man shm_unlink«

### Der Autor

Dr. Willi Nüßer ist Heinz-Nixdorf-Stiftungsprofessor für Informatik an der Fachhochschule der Wirtschaft (FHDW) in Paderborn. Davor war er sechs Jahre lang bei der SAP AG und dort zuletzt als Entwickler im SAP Linuxlab zuständig für die Portierung der SAP-Speicherverwaltung auf Linux.