Technische Universität Braunschweig

Institut für Betriebssysteme und Rechnerverbund

Betriebssysteme – Übung

5R: Speicherverwaltung

Signe Rüsch, Wintersemester 2018

Übersicht

- Prozess-Speichersegmente
- my_malloc
- my_free

Übersicht

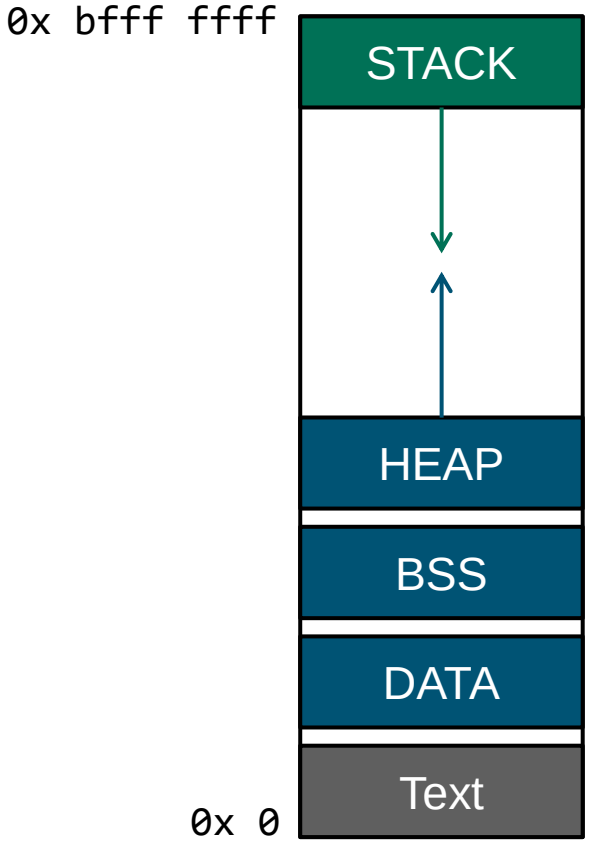
- **Prozess-Speichersegmente**
- my_malloc
- my_free

Prozess-Speichersegmente

Jeder Prozess hat einen virtuellen Adressraum mit mehreren Segmenten

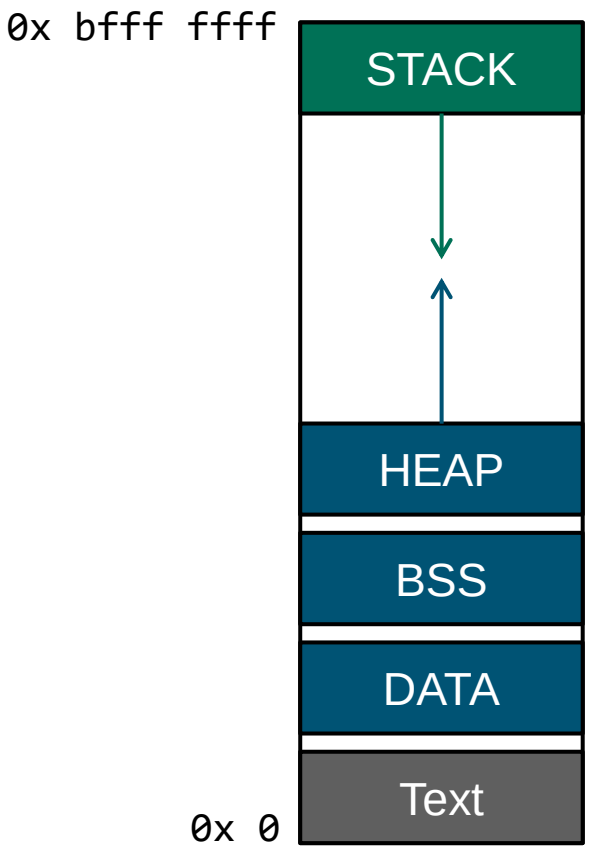
Die wichtigsten sind:

- TEXT
- DATA
- BSS
- HEAP
- STACK



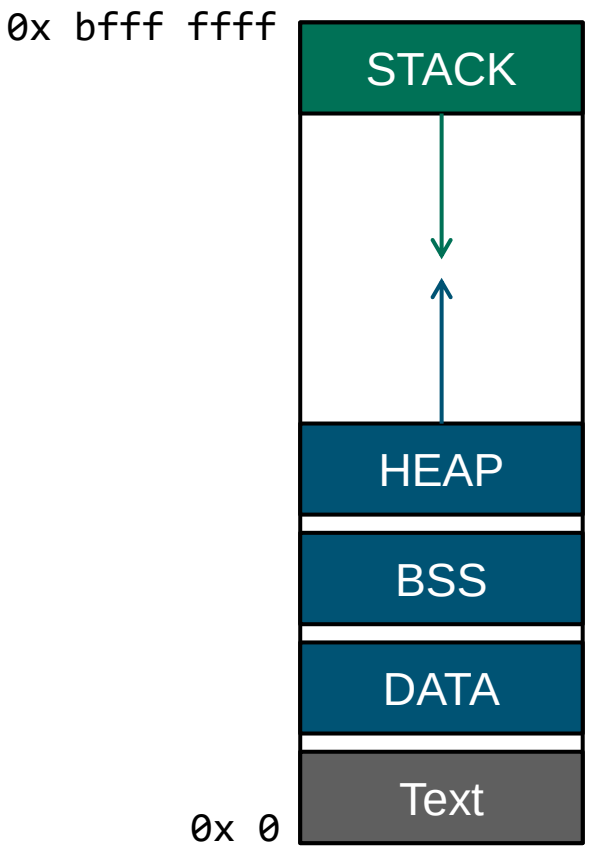
TEXT

- Rechte: lesen/ausführen
- feste Größe
- enthält:
  - Programmcode
  - konstante primitive Datentypen (immediate Werte)
- Abbildung im Hauptspeicher kann von mehreren Prozessen gleichzeitig gelesen werden



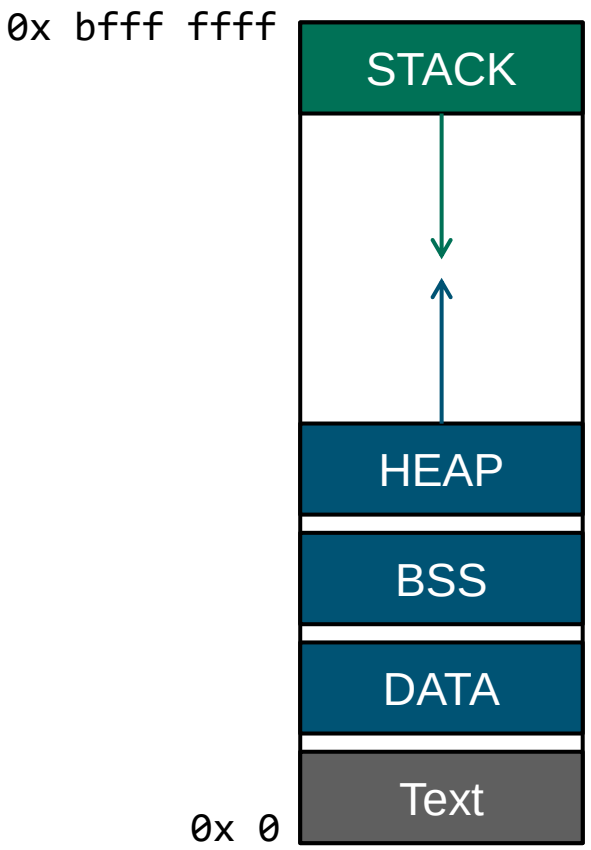
DATA

- Rechte: lesen/schreiben
- feste Größe
- enthält:
  - initialisierte globale Variablen
  - initialisierte statische Variablen
  - Konstanten



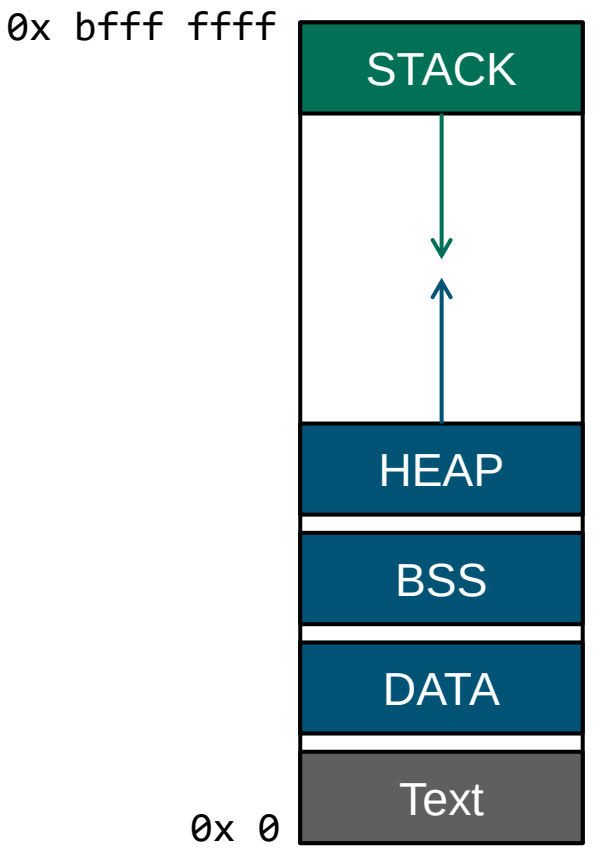
BSS (Block Started by Symbol)

- Rechte: lesen/(schreiben)
- feste Größe
- enthält:
  - nicht initialisierte globale Variablen
  - nicht initialisierte statische Variablen
- Realisierung:
  - technisch zeigen alle Variablen auf eine statische zero-page
  - ➔ initialisiert alle Variablen mit 0
  - bei Schreibzugriff wird die Variable kopiert (Copy-on-write)



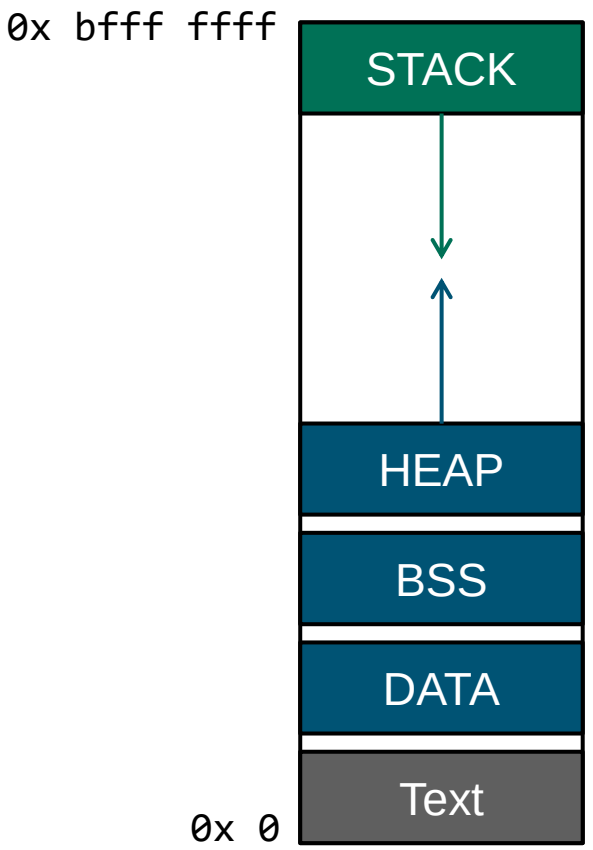
HEAP

- Rechte: lesen/schreiben
- variable Größe
  - ➔ wächst von unten nach oben
- enthält: dynamisch allozierte Daten
- Beispiel für Speicherallokation:
  - malloc
  - new



STACK

- Rechte: lesen/schreiben
- variable Größe
  - ➔ wächst von oben nach unten
- enthält:
  - Argumente für Funktionen
  - lokale Variablen
  - Rücksprungadressen



Beispiel

```c
#include <stdlib.h>
int* field;
unsigned int const size = 10;

int main()
{
  unsigned int bytes = size * sizeof(int);
  field = malloc(bytes);
  for(int i=0; i<size; i++)
  {
    field[i]=i;
  }
  field[6] = 123;
  return field[6];
}
```

Beispiel

```
#include <stdlib.h>
int* field;⇐ field in BSS global, nicht initialisiert
unsigned int const size = 10;

int main()
{
  unsigned int bytes = size * sizeof(int);
  field = malloc(bytes);
  for(int i=0; i<size; i++)
  {
    field[i]=i;
  }
  field[6] = 123;
  return field[6];
}
```

Beispiel

```c
#include <stdlib.h>
int* field;
unsigned int const size = 10;

int main()
{
  unsigned int bytes = size * sizeof(int);
  field = malloc(bytes);
  for(int i=0; i<size; i++)
  {
    field[i]=i;
  }
  field[6] = 123;
  return field[6];
}
```

⇐ **field** in **BSS** global, nicht initialisiert

⇐ **size** in **DATA** global, initialisiert

Beispiel

```c
#include <stdlib.h>
int* field;
unsigned int const size = 10;

int main()
{
  unsigned int bytes = size * sizeof(int);
  field = malloc(bytes);
  for(int i=0; i<size; i++)
  {
    field[i]=i;
  }
  field[6] = 123;
  return field[6];
}
```

⇐ **field** in **BSS** global, nicht initialisiert

⇐ **size** in **DATA** global, initialisiert

⇐ **main** in **TEXT** Programmcode

Beispiel

```c
#include <stdlib.h>
int* field;
unsigned int const size = 10;

int main()
{
  unsigned int bytes = size * sizeof(int);
  field = malloc(bytes);
  for(int i=0; i<size; i++)
  {
    field[i]=i;
  }
  field[6] = 123;
  return field[6];
}
```

⇐ **field** in **BSS** global, nicht initialisiert

⇐ **size** in **DATA** global, initialisiert

⇐ **main** in **TEXT** Programmcode

⇐ **bytes** in **STACK** lokale Variable

Beispiel

```c
#include <stdlib.h>
int* field;
unsigned int const size = 10;

int main()
{
  unsigned int bytes = size * sizeof(int);
  field = malloc(bytes);
  for(int i=0; i<size; i++)
  {
    field[i]=i;
  }
  field[6] = 123;
  return field[6];
}
```

⇐ **field** in **BSS** global, nicht initialisiert

⇐ **size** in **DATA** global, initialisiert

⇐ **main** in **TEXT** Programmcode

⇐ **bytes** in **STACK** lokale Variable

⇐ **i** in **STACK** lokale Variable

Beispiel

```c
#include <stdlib.h>
int* field;
unsigned int const size = 10;

int main()
{
  unsigned int bytes = size * sizeof(int);
  field = malloc(bytes);
  for(int i=0; i<size; i++)
  {
    field[i]=i;
  }
  field[6] = 123;
  return field[6];
}
```

⇐ **field** in **BSS** global, nicht initialisiert

⇐ **size** in **DATA** global, initialisiert

⇐ **main** in **TEXT** Programmcode

⇐ **bytes** in **STACK** lokale Variable

⇐ **i** in **STACK** lokale Variable

⇐ **field** zeigt in **HEAP** Pointer field zeigt auf HEAP

Beispiel

```c
#include <stdlib.h>
int* field;
unsigned int const size = 10;

int main()
{
  unsigned int bytes = size * sizeof(int);
  field = malloc(bytes);
  for(int i=0; i<size; i++)
  {
    field[i]=i;
  }
  field[6] = 123;
  return field[6];
}
```

⇐ **field** in **BSS** global, nicht initialisiert

⇐ **size** in **DATA** global, initialisiert

⇐ **main** in **TEXT** Programmcode

⇐ **bytes** in **STACK** lokale Variable

⇐ **i** in **STACK** lokale Variable

⇐ **field** zeigt in **HEAP** Pointer field zeigt auf HEAP

⇐ **123** in **TEXT** immediate-Wert ↪ Programmcode

Übersicht

- Prozess-Speichersegmente
- **my_malloc**
- my_free

Einfache malloc-Implementierung (my_malloc)

Ziel der Aufgabe

- Zusammenhang zwischen „nacktem Speicher“ und typisierten Datenbereichen verstehen
- Funktionen aus der C-Bibliothek selbst realisieren

Vereinfachungen:

- First-Fit-ähnliche Allokationsstrategie
- nur eine Page dynamisch alloziert via mmap
- freier Speicher wird in einer doppelt-Verketteten Liste verwaltet

Verwaltungsstrukturen

Zweigliedrige Verwaltung bei Speicher Anfragen:

- Erste Speicher Anforderung: Verwaltung durch BS
  - neue Page allozieren
  - Mapping des Speichers in Programm Adressraum via mmap
- weitere Speicheranforderung: Verwaltung von Memoryblocks
  - eigene Verwaltungsstruktur
  - im wesentlichen doppelt-verkettete Liste der freien Speicherbereiche
  - benutzte Speicherbereiche nicht Teil der zu verwaltenden Liste

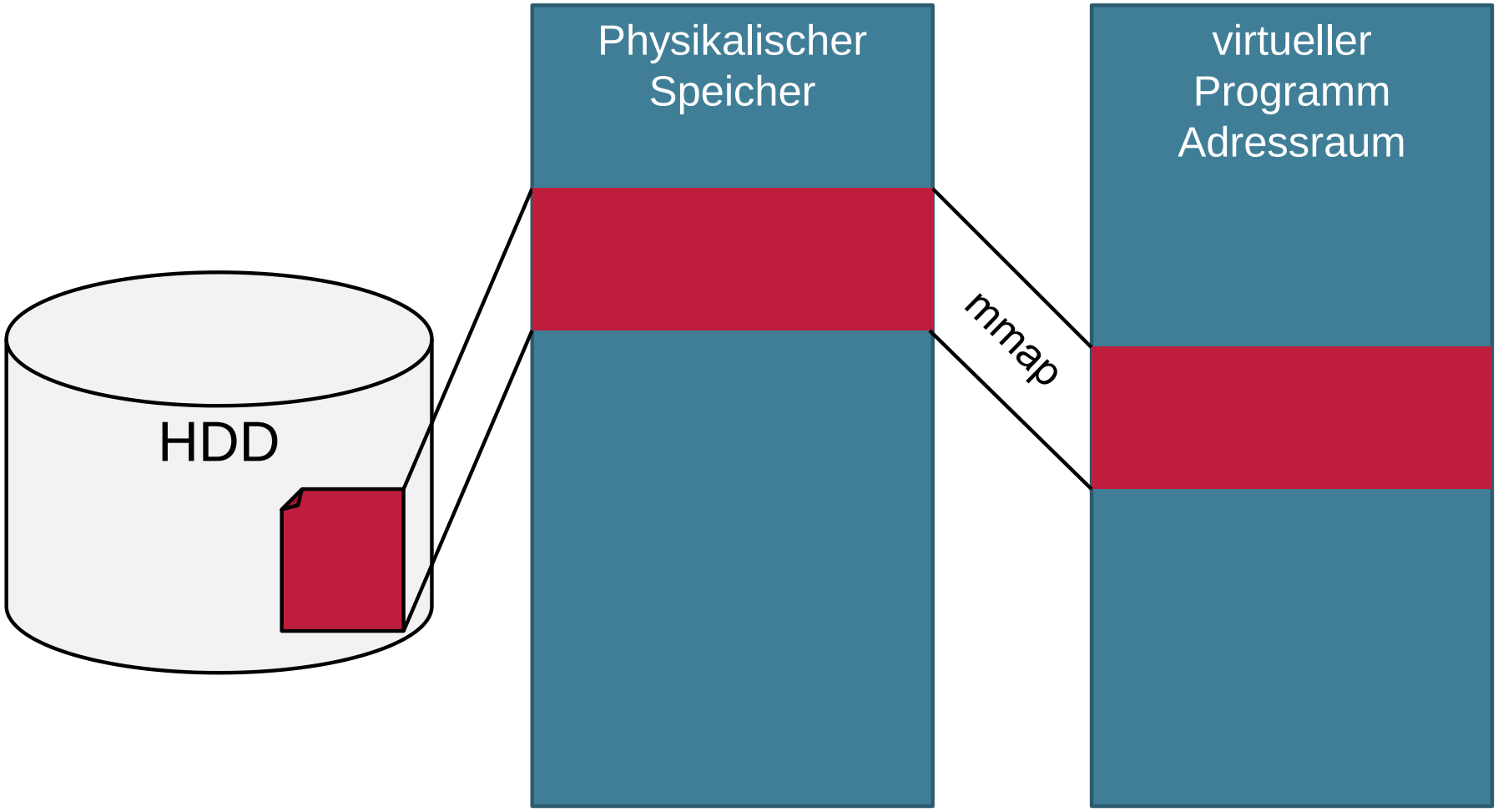
Exkurs mmap

Was ist mmap?

- Systemcall zum Kernel
- alloziert Speicher auf Page-Ebene
- kann Dateien direkt in den Programm-Adressraum einbinden

```
void *mmap(
  void *addr,     // Adresse wohin Speicher gemappt wird
  size_t length, // Laenge des allozierenden Speichers
  int prot,       // Protokoll (Lesen/Schreiben/Ausfuehren etc.)
  int flags,      // Anonymous Mapping, Shared Memory etc.
  int fd,                 // File Descriptor der einzubindenen Datei
  off_t offset    // Speicher offset in der Datei
  );
```

Datei Mapping Funktionsweise



Beispiel Anonymous mapping

```
...
int *memory = NULL;
//anonymous mapping
memory=mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1, 0);

//error detection and handling
if(memory == MAP_FAILED){
  perror("Memory Mapping Error");
  exit(-1);
}

//initialize memory for further use
int *temp = memory;
for(unsigned int i = 0; i < 4096/sizeof(int); ++i){
  *(temp++) = 0;
}
...
```

my_malloc-Funktion

- Verwaltet folgende Informationen über einen Speicherbereich
  - welche Pages wurden vergeben
  - welche Bereiche in einer Page sind frei

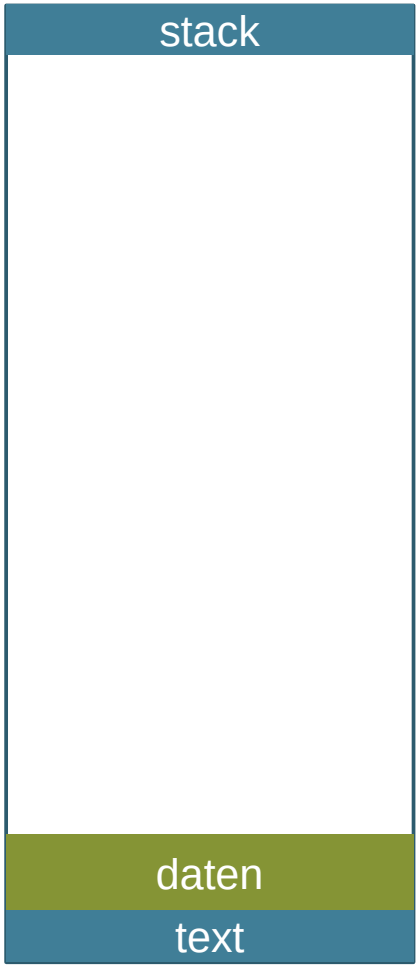
```
memblock_t;
chunk_t;
```

- chunk_t verwaltet Informationen über Pages
  - welche Datei gehört zu welcher Page
  - wie groß ist der Speicherbereich
  - wo liegt der erste freie Memory Block
- memblock_t enthält Information von freien Speicherbereichen
  - Größe des Speicherbereichs
  - vorherige / kommende freie Speicherbereiche

my_malloc-Interna: Initialisierung

Initialer Zustand

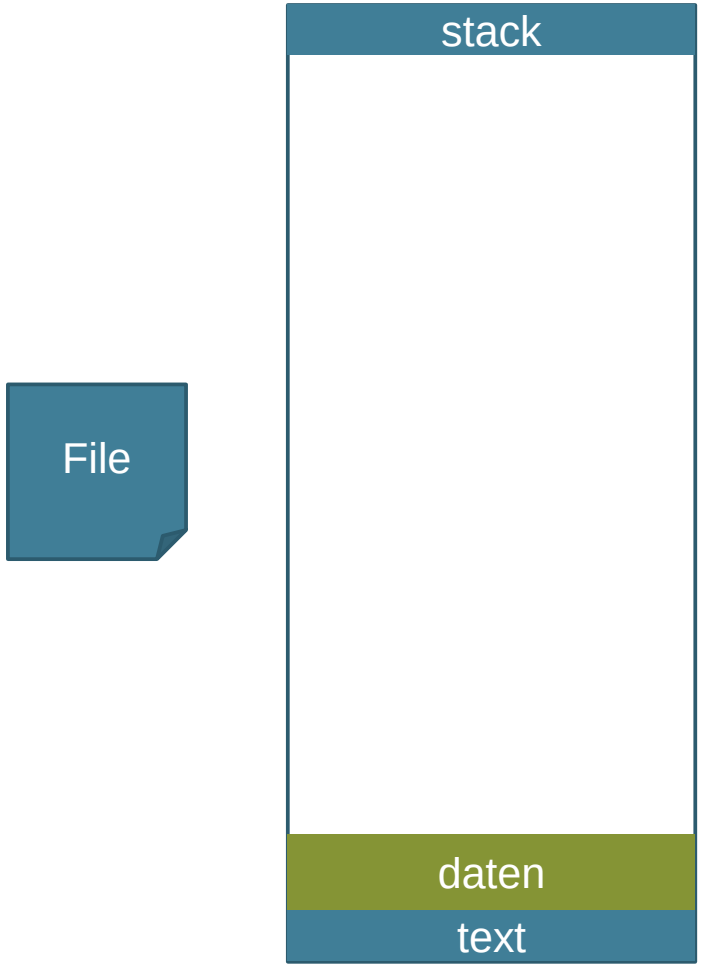
- Kein Speicher alloziert



my_malloc-Interna: Initialisierung

Initialer Zustand

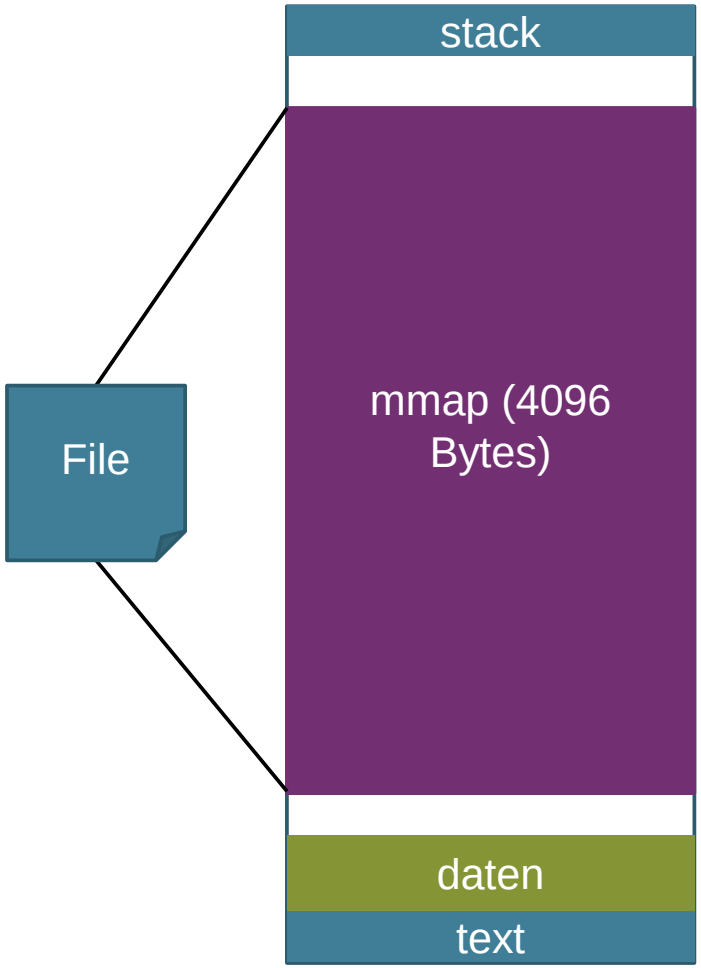
- Kein Speicher alloziert
- Datei anlegen
- Speicher reservieren



my_malloc-Interna: Initialisierung

Initialer Zustand

- Kein Speicher alloziert
- Datei anlegen
- Speicher reservieren
- mmaps



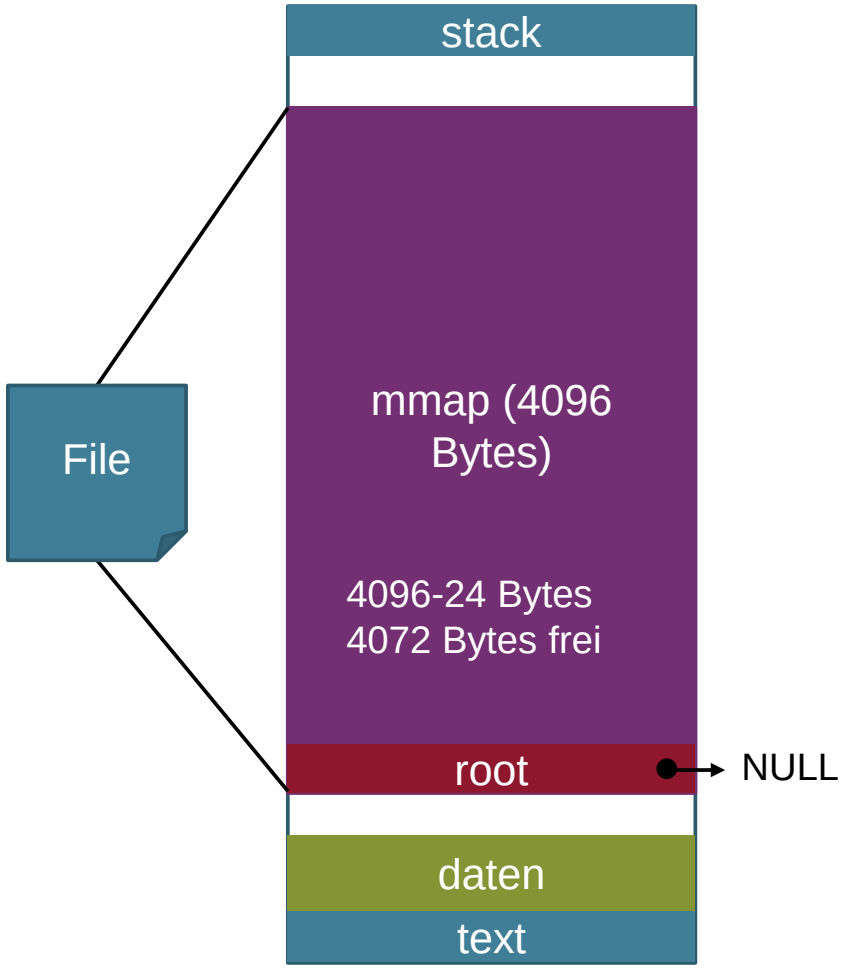
my_malloc-Interna: Initialisierung

Initialer Zustand

- Kein Speicher alloziert
- Datei anlegen
- Speicher reservieren
- mmaps
- Anlegen des Wurzelknotens

chunk_t und memblock_t haben jeweils die Größe 24 Byte

In der Rechnerübung sizeof anstelle von absoluten Zahlen nutzen

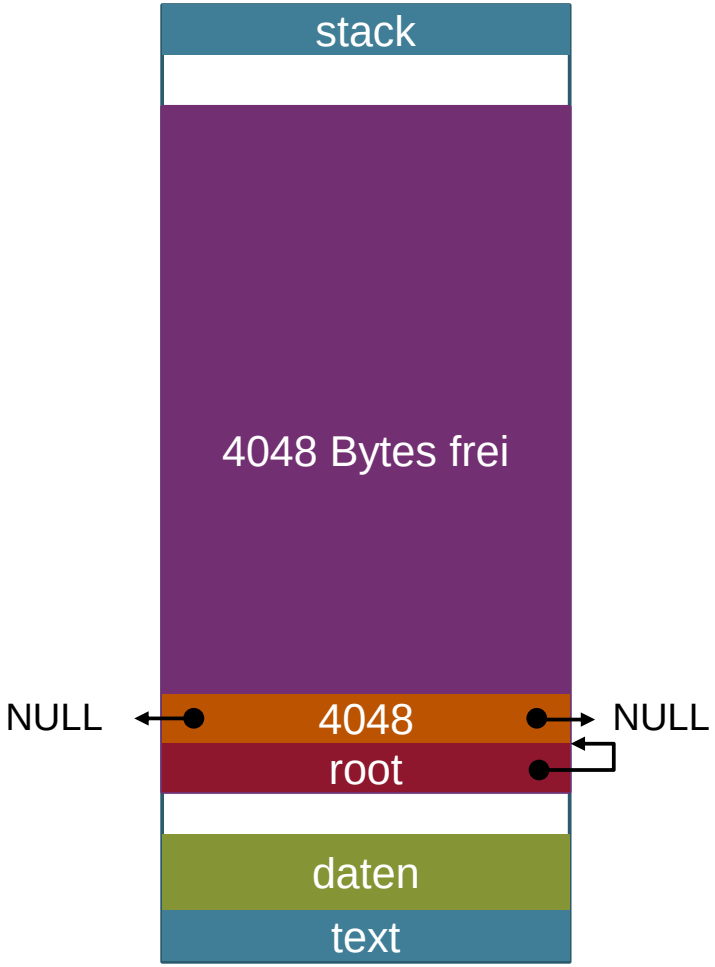


chunk_t
memblock_t
mmap

my_malloc-Interna: Allokation von Speicher

Drei Speicherbereiche reservieren:

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
```



my_malloc-Interna: Allokation von Speicher

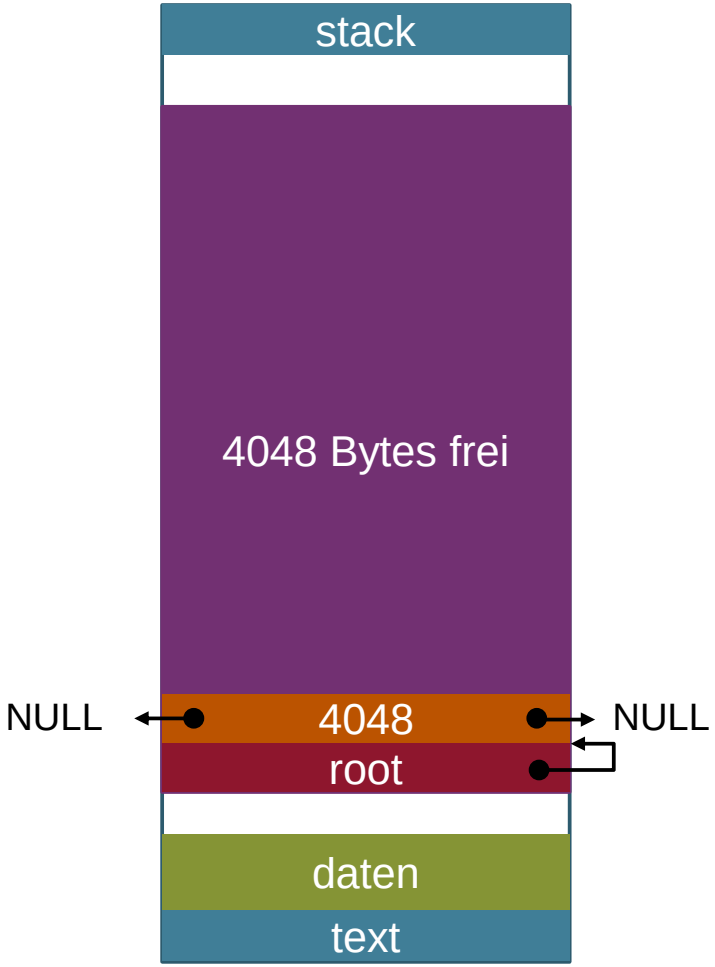
chunk_t
memblock_t
mmap

Drei Speicherbereiche reservieren:

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
```

- runden des zu reservierenden Speichers

```
malloc(size_t size){ // size=128
...
--size;                    // size=127
size /= sizeof(memblock_t);//size=5
++size;                    // size=6
size *= sizeof(memblock_t);//size=144
...
}
```



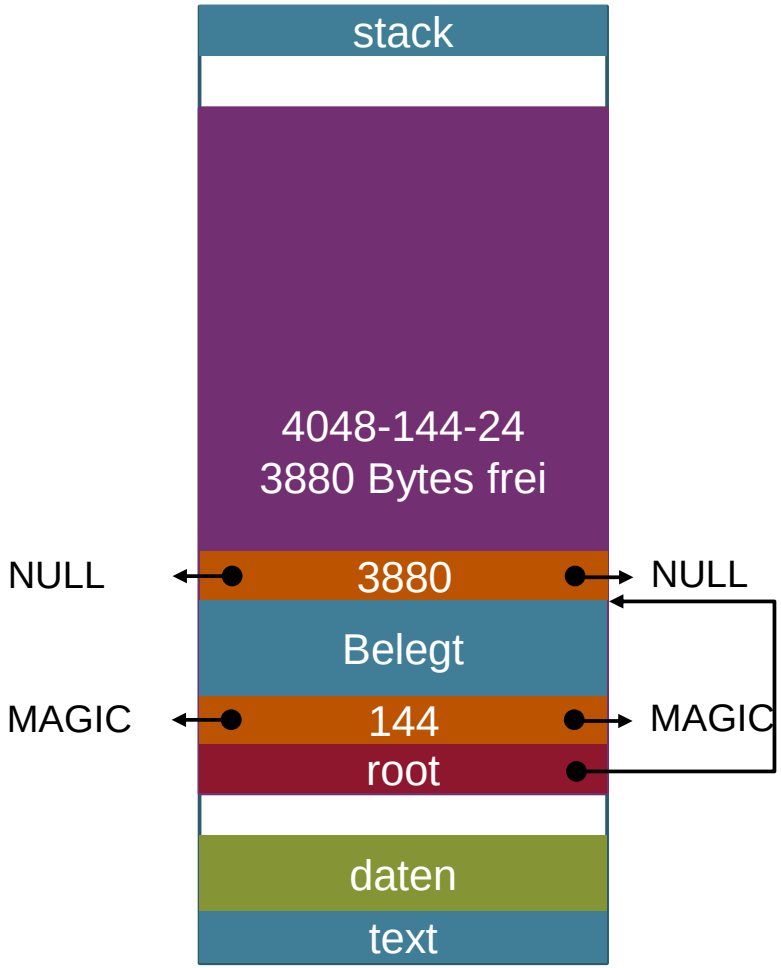
chunk_t
memblock_t
mmap

my_malloc-Interna: Allokation von Speicher

Drei Speicherbereiche reservieren:

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
```

- runden des zu reservierenden Speichers
- Zeiger auf MAGIC zeigen lassen
- neuen memblock anlegen
- neuen freien Speicher berechnen

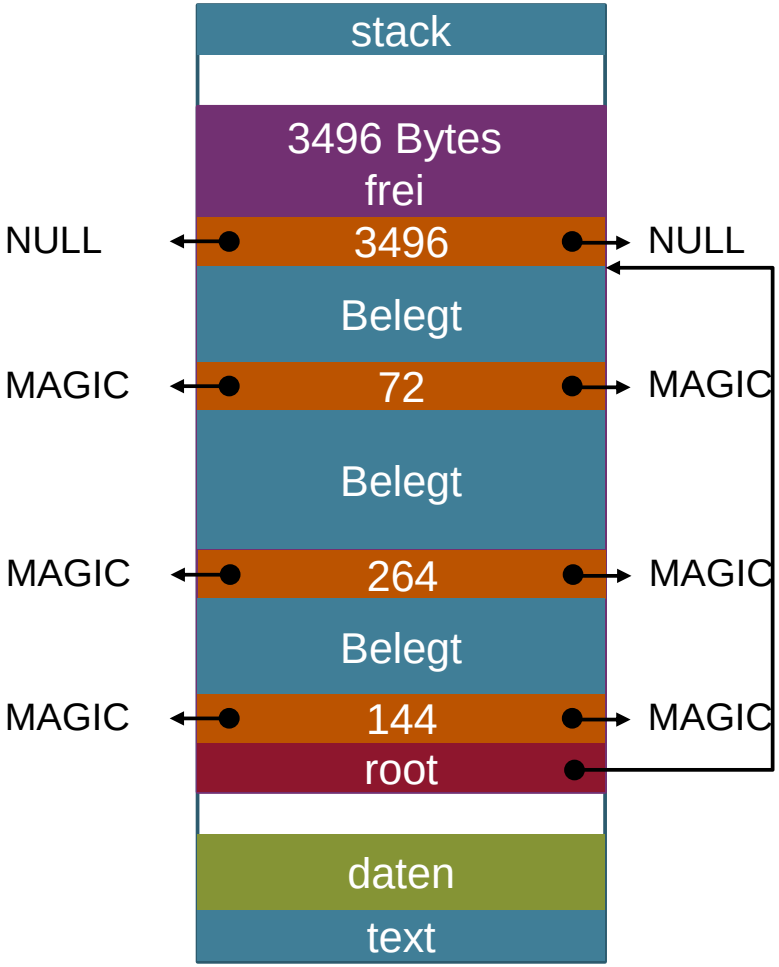


chunk_t
memblock_t
mmap

my_malloc-Interna: Allokation von Speicher

Drei Speicherbereiche reservieren:

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
```

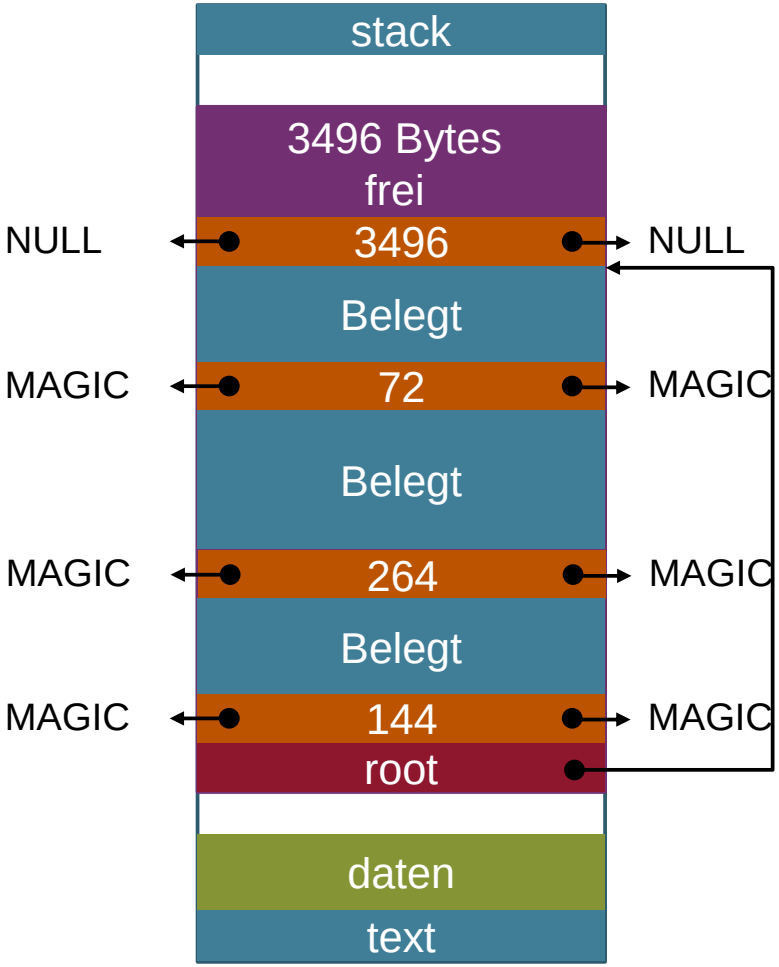


my_malloc-Interna: Allokation von Speicher

chunk_t
memblock_t
mmap

Drei Speicherbereiche reservieren:

```
char *a, *b, *c, *d;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
d= (char *) malloc(240);
```

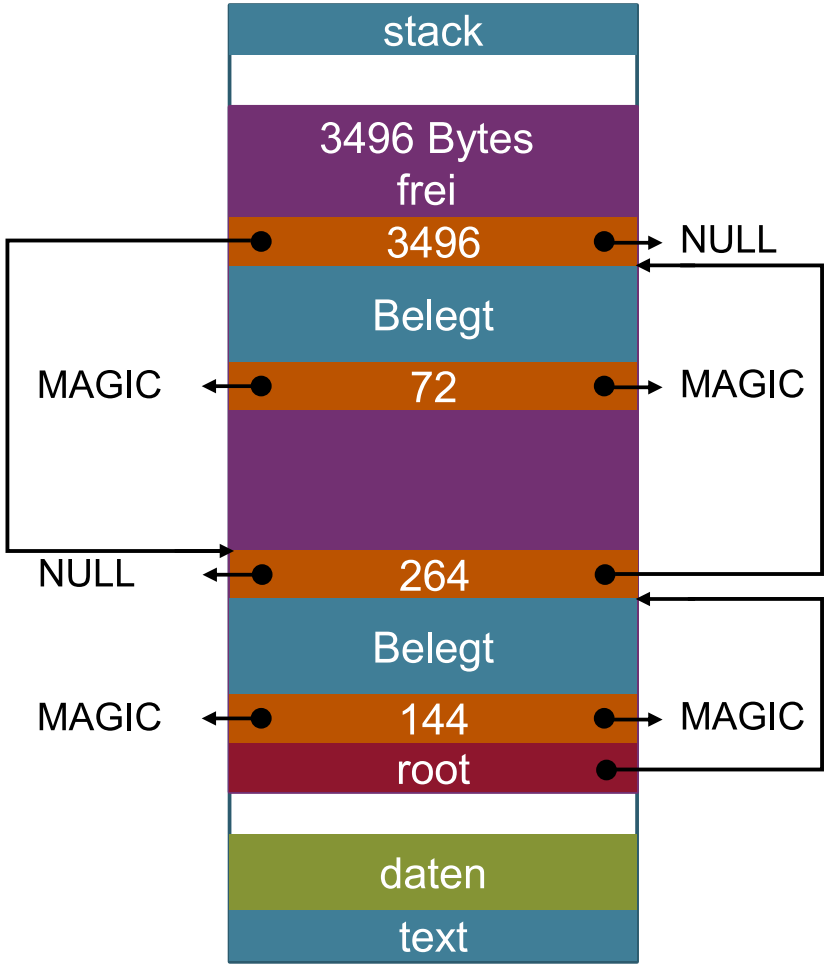


my_malloc-Interna: Allokation von Speicher

chunk_t
memblock_t
mmap

Drei Speicherbereiche reservieren:

```
char *a, *b, *c, *d;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
d= (char *) malloc(240);
```



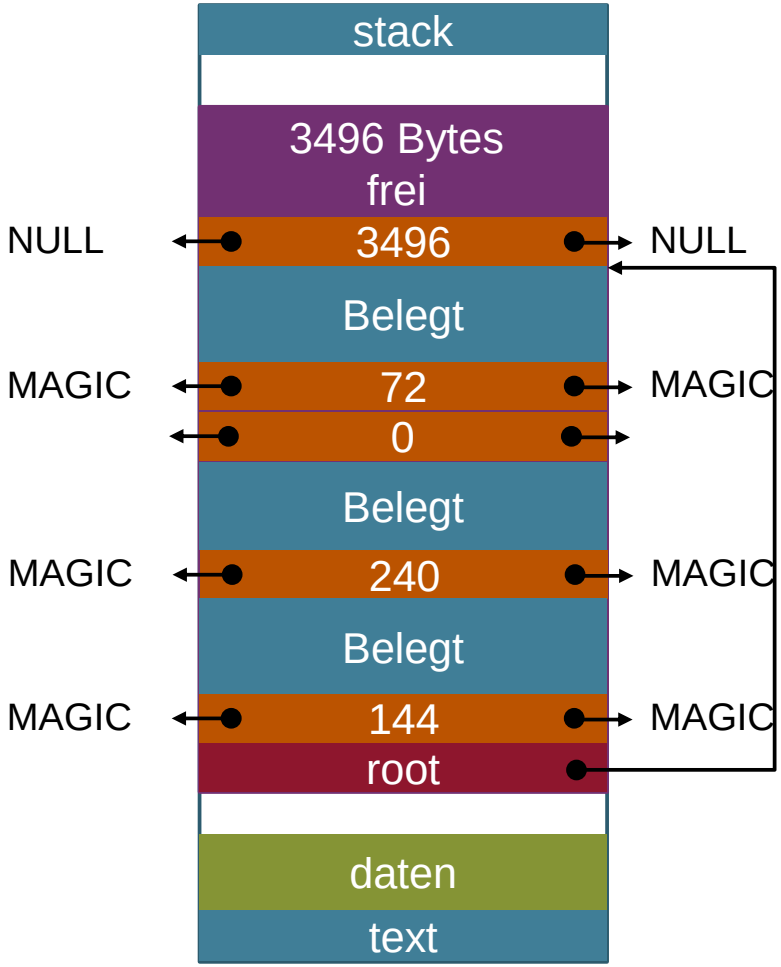
my_malloc-Interna: Allokation von Speicher

chunk_t
memblock_t
mmap

Drei Speicherbereiche reservieren:

```
char *a, *b, *c, *d;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
d= (char *) malloc(240);
```

Memblocks der Größe 0 sollen nicht angelegt werden!



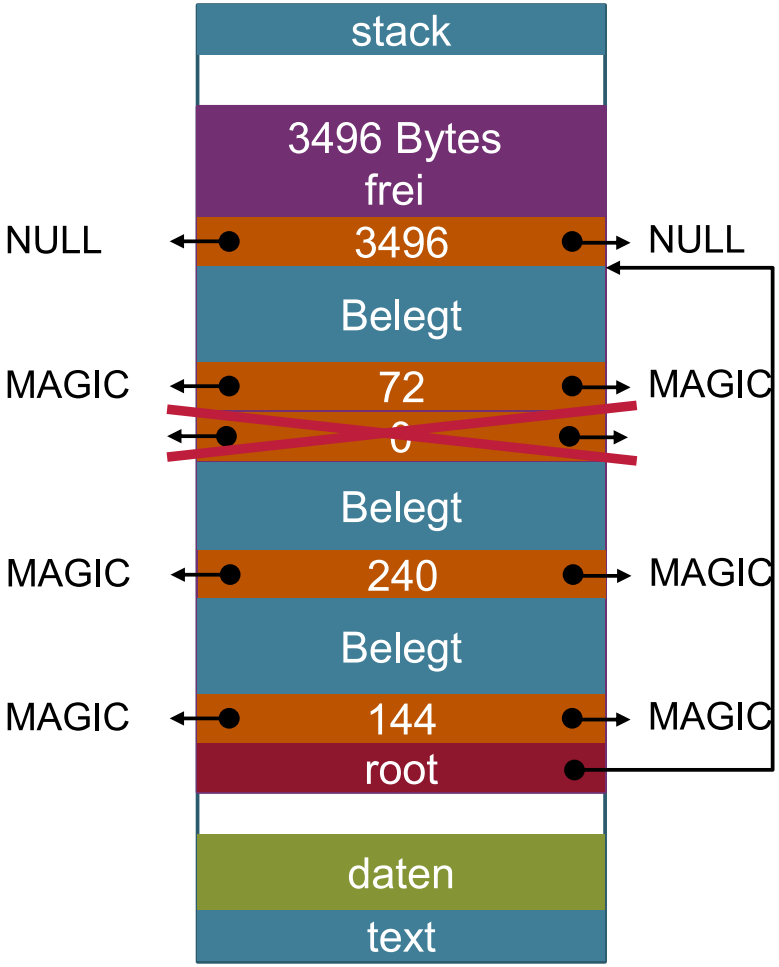
chunk_t
memblock_t
mmap

my_malloc-Interna: Allokation von Speicher

Drei Speicherbereiche reservieren:

```
char *a, *b, *c, *d;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
d= (char *) malloc(240);
```

Memblocks der Größe 0 sollen nicht angelegt werden!



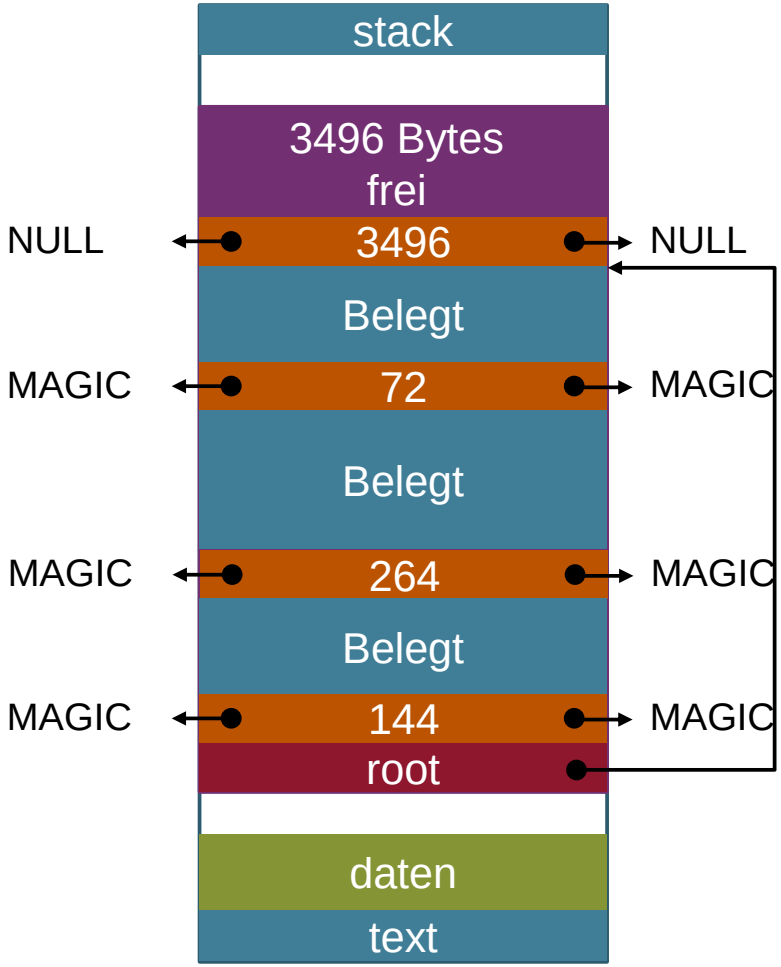
chunk_t
memblock_t
mmap

my_malloc-Interna: Allokation von Speicher

Drei Speicherbereiche reservieren:

```
char *a, *b, *c, *d;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
d= (char *) malloc(240);
```

Memblocks der Größe 0 sollen nicht angelegt werden!



Übersicht

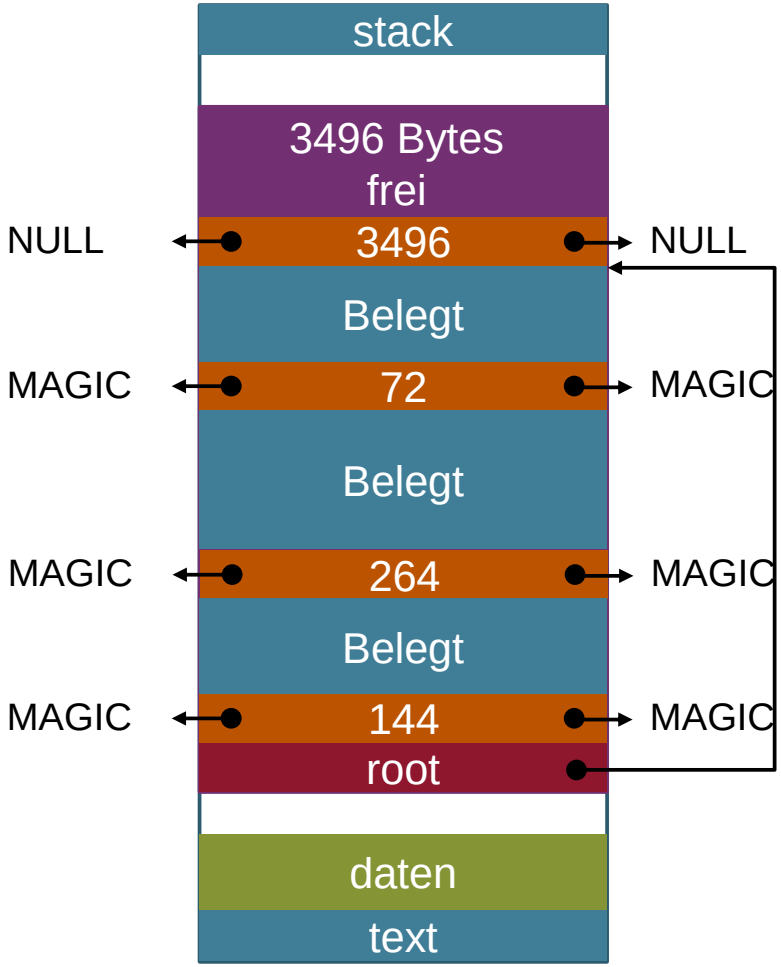
- Prozess-Speichersegmente
- my_malloc
- **my_free**

chunk_t
memblock_t
mmap

my_free-Funktion: Speicher freigeben

Situation nach drei my_malloc-Aufrufen:

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
```

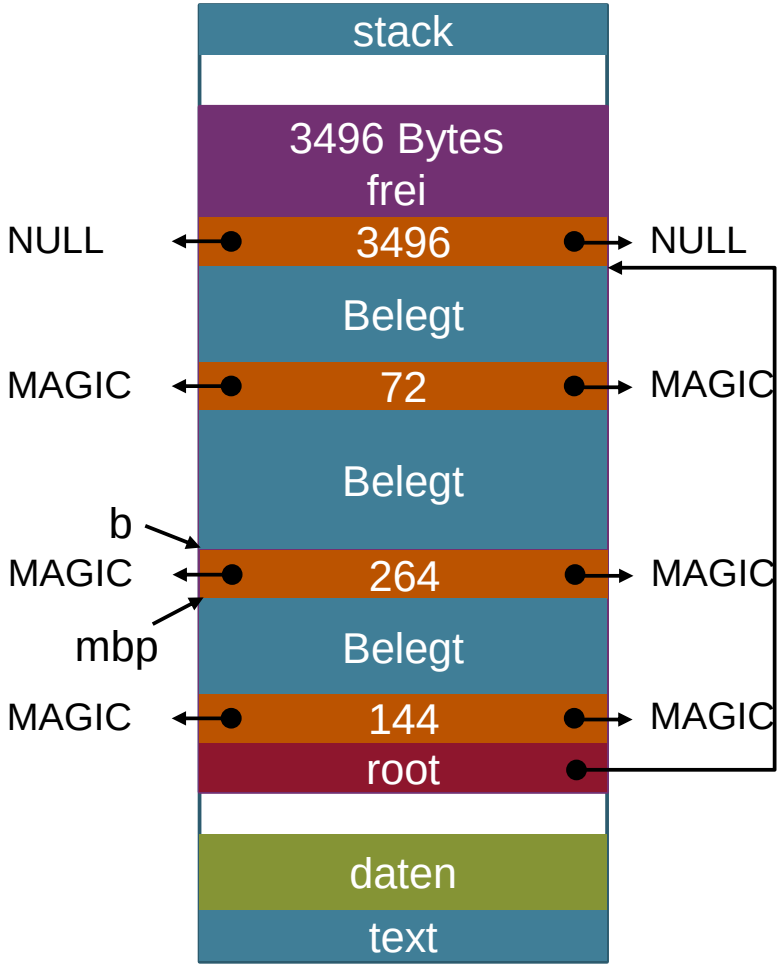


my_free-Funktion: Speicher freigeben

Freigabe von b

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
```

- Zeiger auf mbp zugehörigen memblock ermitteln
- überprüfen, ob ein gültiger, belegter memblock vorliegt (MAGIC)

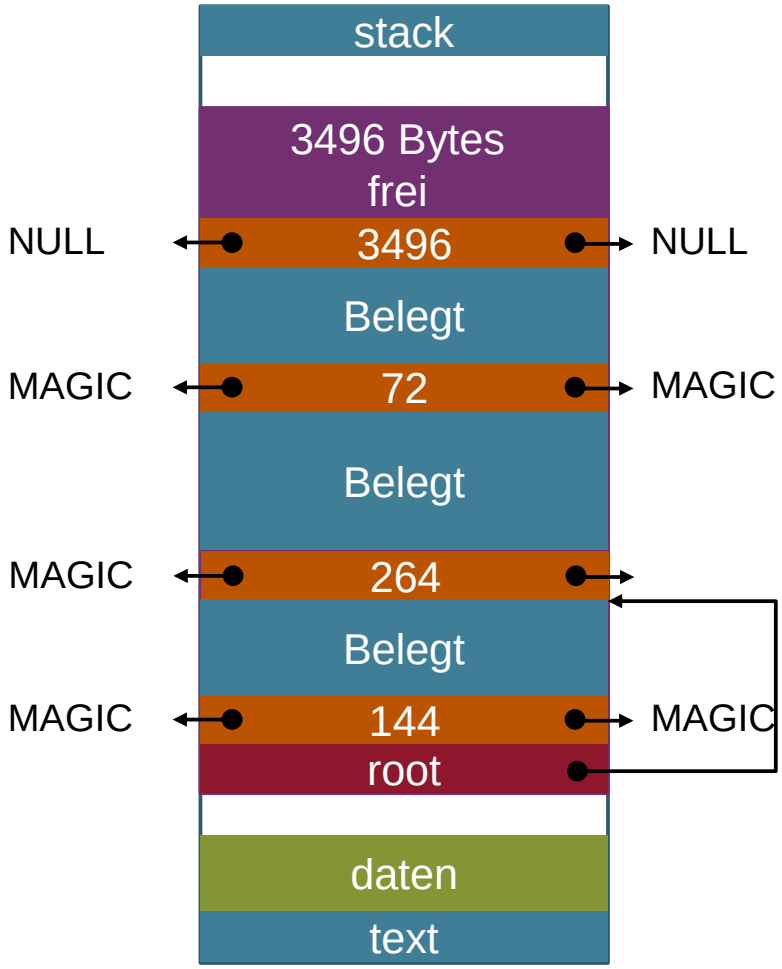


my_free-Funktion: Speicher freigeben

Freigabe von b

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
```

- Zeiger auf mbp zugehörigen memblock ermitteln
- überprüfen, ob ein gültiger, belegter memblock vorliegt (MAGIC)
- Einhängen des Blocks

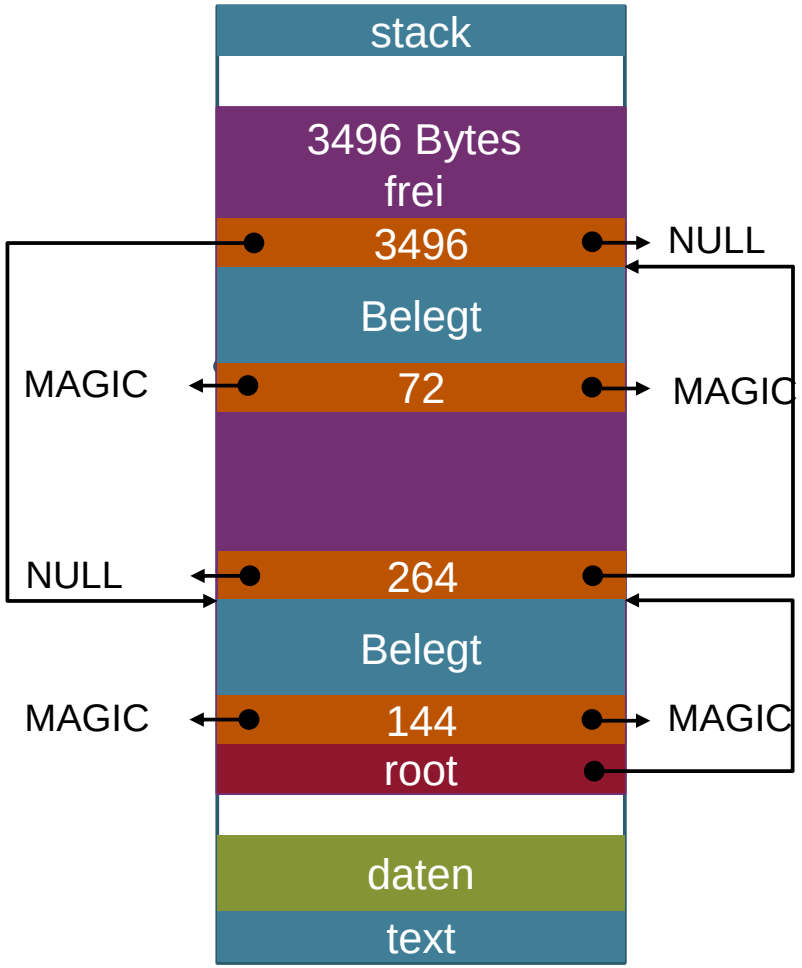


my_free-Funktion: Speicher freigeben

Freigabe von b

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
```

- Zeiger auf mbp zugehörigen memblock ermitteln
- überprüfen, ob ein gültiger, belegter memblock vorliegt (MAGIC)
- Einhängen des Blocks

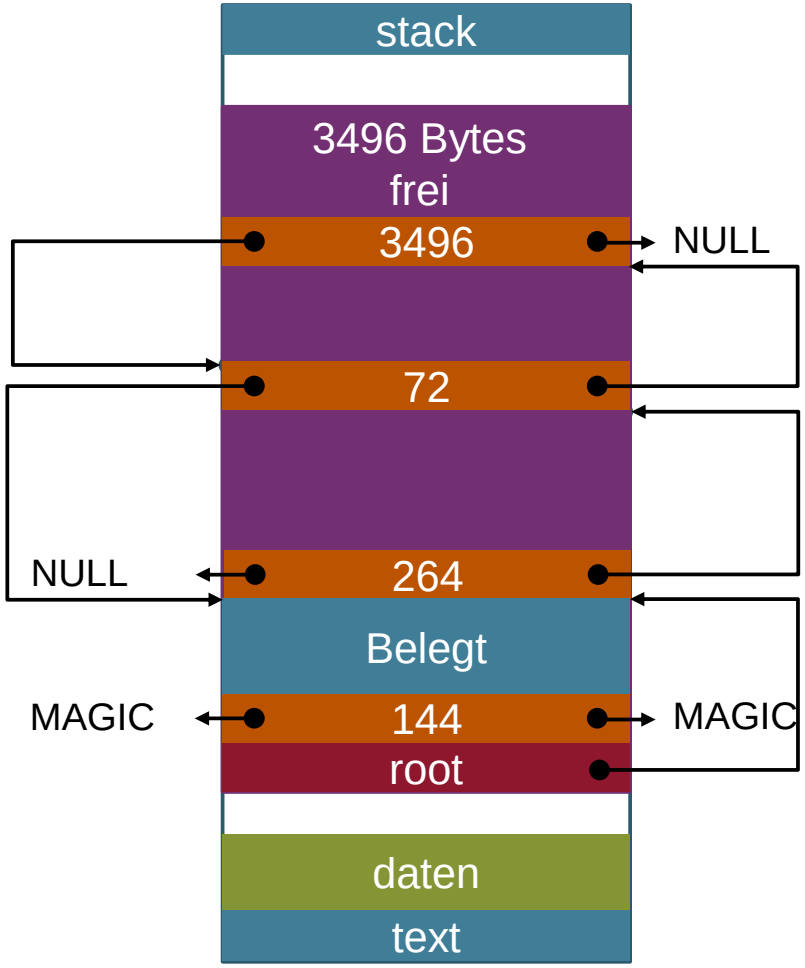


my_free-Funktion: Speicher freigeben

Freigabe von c

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
free(c);
```

- Gleiches Vorgehen wie bei b
- Zusammenfassen der ersten Blöcke
- Neue Größe Berechnen:
  3496 + 72 + 24 = 3592

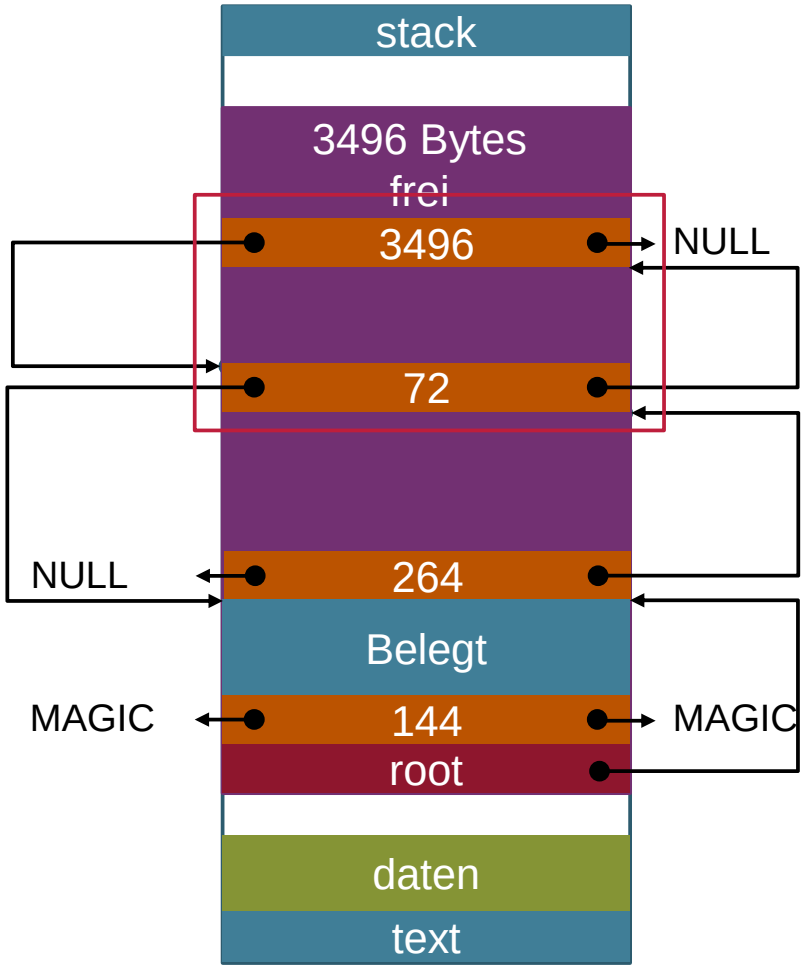


my_free-Funktion: Speicher freigeben

Freigabe von c

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
free(c);
```

- Gleiches Vorgehen wie bei b
- Zusammenfassen der ersten Blöcke
- Neue Größe Berechnen:
  3496 + 72 + 24 = 3592

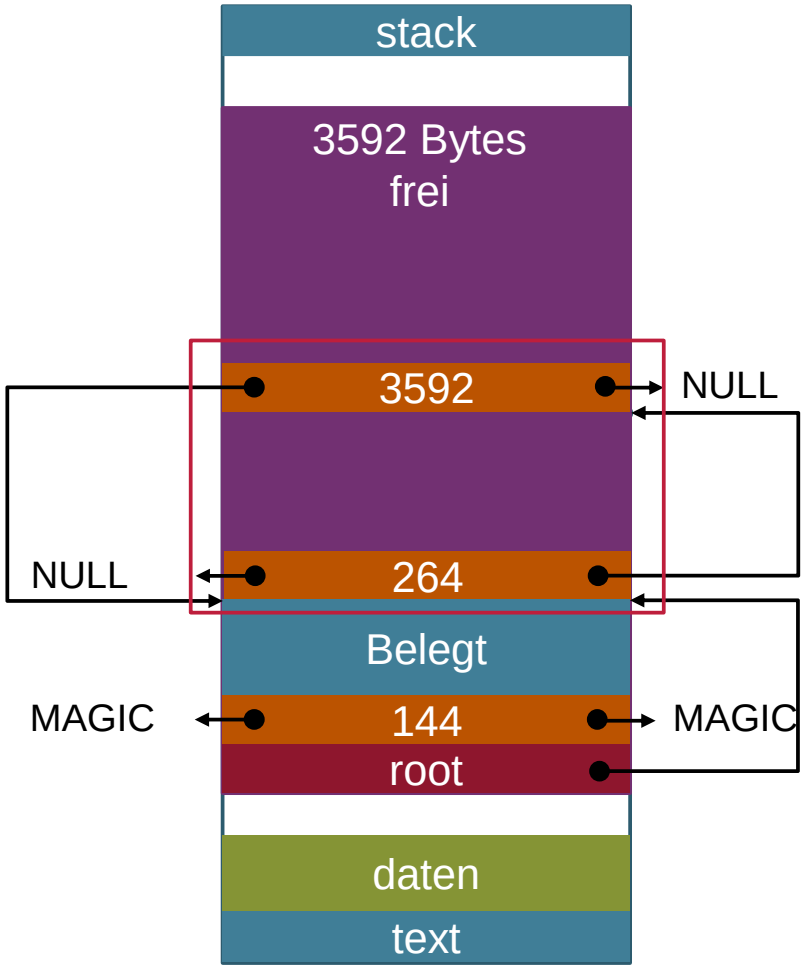


my_free-Funktion: Speicher freigeben

Freigabe von c

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
free(c);
```

- Gleiches Vorgehen wie bei b
- Zusammenfassen der ersten Blöcke
- Neue Größe Berechnen:
  3496 + 72 + 24 = 3592

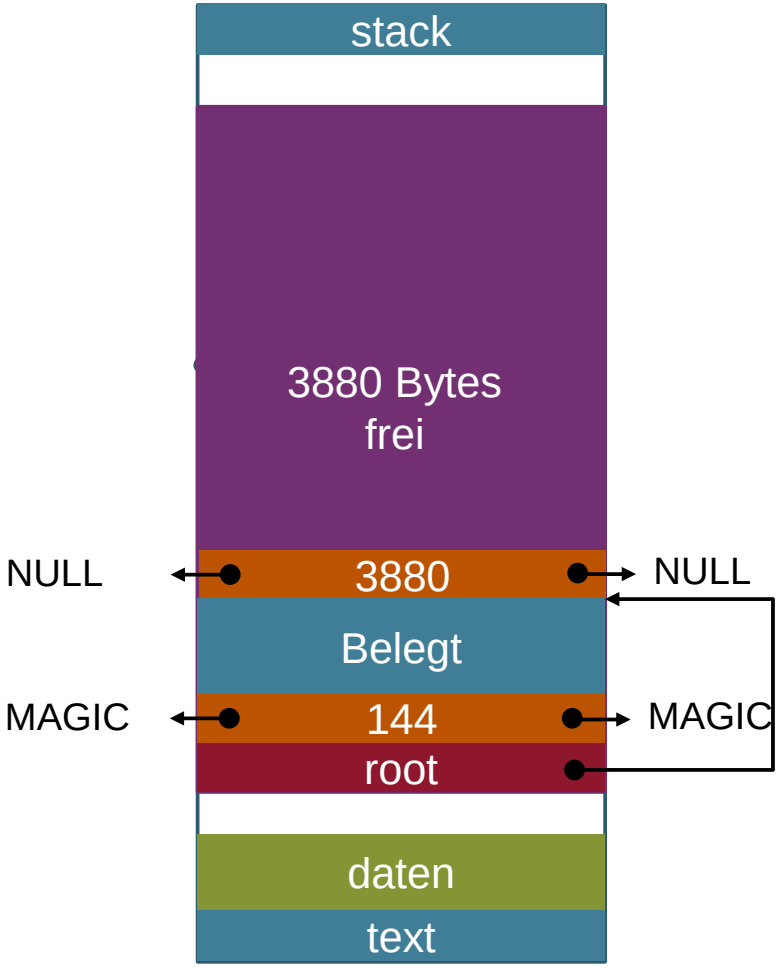


my_free-Funktion: Speicher freigeben

Freigabe von c

```
char *a, *b, *c;
...
a= (char *) malloc(128);
b= (char *) malloc(256);
c= (char *) malloc( 64);
free(b);
free(c);
```

- Gleiches Vorgehen wie bei b
- Zusammenfassen der ersten Blöcke
- Neue Größe Berechnen: 3496 + 72 + 24 = 3592
- Wiederholen bis alle benachbarten Blöcke zusammenhängen



my_malloc-Funktion: Abschließende Bemerkung

Sehr einfache Implementierung - in der Praxis problematisch

- Suche nach passender Lücke kann u.U. länger dauern
  - Lösung: Binäre-Bäume nutzen, hierarchische Allokation
- hoher Verwaltungsoverhead
  - Doppelt-Verkettete Listen leicht zu implementieren
  - Lösung: Nutzung von Bit-Arrays
- Sinnvolle Implementierung erfordert geeignete Speichervergabestrategie
  - Implementierung erheblich Aufwändiger – Resultat aber entsprechend effizienter
  - Strategien werden in der Vorlesung behandelt

Vielen Dank für Ihre Aufmerksamkeit!