

U4 4. Übung

U4-1 Überblick

- Aufgabe 2: qsort - Fortsetzung
- Infos zur Aufgabe 4: malloc-Implementierung

U4-2 Aufgabe 2: Sortieren mittels qsort (Fortsetzung)

1 wsort - Datenstrukturen (1. Möglichkeit)

- Array von Zeichenketten

H	a	l	l	o	\0	...	\0	T	e	s	t	\0	\0	...	\0	H	a	n	s	...
0						...		100	101					...		201	202			

- Vorteile:

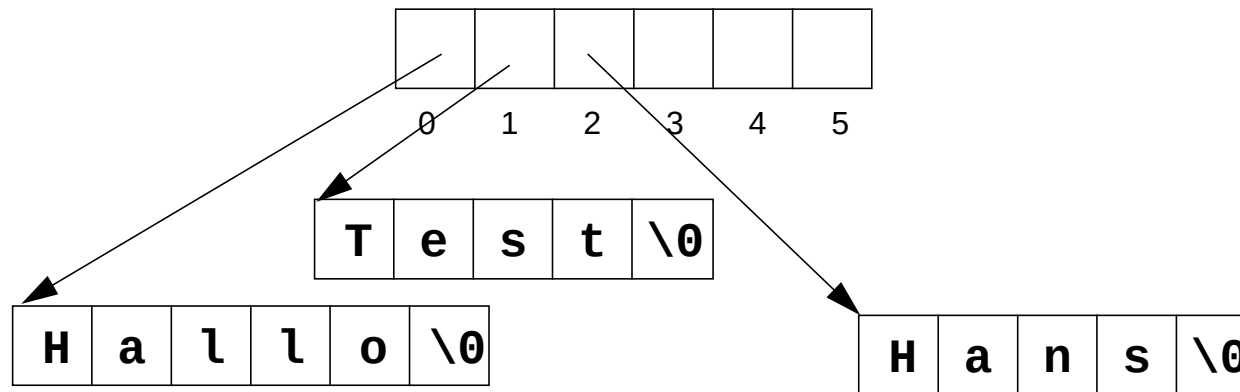
- ◆ einfach

- Nachteile:

- ◆ hoher Kopieraufwand
- ◆ Maximale Länge der Worte muss bekannt sein
- ◆ Verschwendung von Speicherplatz

2 wsort - Datenstrukturen (2. Möglichkeit)

■ Array von Zeigern auf Zeichenketten



■ Vorteile:

- ◆ schnelles Sortieren, da nur Zeiger vertauscht werden müssen
- ◆ Zeichenketten können beliebig lang sein
- ◆ sparsame Speichernutzung

3 Speicherverwaltung

- Berechnung des Array-Speicherbedarfs
 - ◆ bei Lösung 1: Anzahl der Wörter * 101 * sizeof(char)
 - ◆ bei Lösung 2: Anzahl der Wörter * sizeof(char*)
- realloc:
 - ◆ Anzahl der zu lesenden Worte ist unbekannt
 - ◆ Array muß vergrößert werden: realloc
 - ◆ Bei Vergrößerung sollte man aus Effizienzgründen nicht nur Platz für ein neues Wort (Lösungsvariante 1) bzw. einen neuen Zeiger (Lösungsvariante 2) besorgen, sondern für mehrere.
 - ◆ Achtung: realloc kopiert möglicherweise das Array (teuer)
- Speicher sollte wieder freigegeben werden
 - ◆ bei Lösung 1: Array freigeben
 - ◆ bei Lösung 2: zuerst Wörter freigeben, dann Zeiger-Array freigeben

4 Vergleichsfunktion

- Problem: qsort erwartet folgenden Funktionszeigertyp:

```
int (*)(const void *, const void *)
```

- Lösung: "casten"

- ◆ innerhalb der Funktion, z.B. (Feld vom Typ char **):

```
int compare(const void *a, const void *b) {
    return strcmp(*(char **)a, *((char **)b));
}
```

- ◆ beim qsort-Aufruf:

```
int compare(char **a, char **b);
...
qsort(    field, nel, sizeof(char *),
        (int (*)(const void *, const void *))compare);
```

U4-3 Aufgabe 4: einfache malloc-Implementierung

1 Überblick

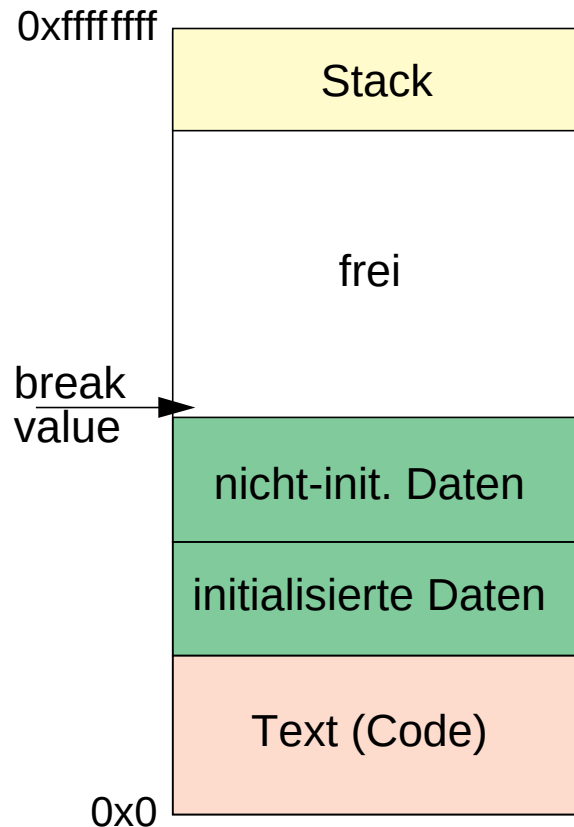
- erheblich vereinfachte Implementierung
 - nur einmal am Anfang Speicher vom Betriebssystem anfordern (1 MB)
 - freigegebener Speicher wird in einer einfachen verketteten Liste verwaltet (benachbarte freie Blöcke werden nicht mehr verschmolzen)
 - **realloc** verlängert den Speicher nicht, sondern wird grundsätzlich auf ein neues **malloc**, **memcpy** und **free** abgebildet

2 Ziele der Aufgabe

- Zusammenhang zwischen "nacktem Speicher" und typisierten Datenbereichen verstehen
- Beispiel für eine Funktion aus einer Standard-Bibliothek erstellen

3 Speicher vom Betriebssystem anfordern

- Speicher im Anschluss an das Datensegment kann vom Betriebssystem angefordert werden



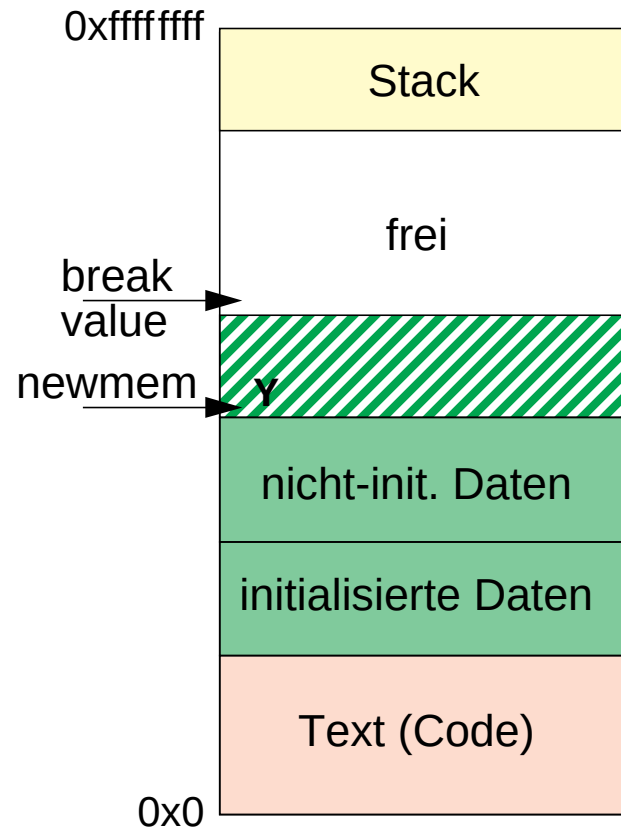
- ◆ break value = Adresse direkt hinter dem Datensegment
- ◆ Systemaufruf erlaubt es, diese Adresse neu festzulegen
 - es entsteht zusätzlicher Speicher hinter den nicht-initialisierten Daten
- ◆ Schnittstellen:

```
int brk(void *endds);  
void *sbrk(intptr_t incr);
```

brk setzt den break value absolut neu fest
sbrk erhöht den break value um incr Bytes

3 Speicher vom Betriebssystem anfordern (2)

- Speicher im Anschluss an das Datensegment kann vom Betriebssystem angefordert werden



◆ Beispiel: 8 KB Speicher anfordern

```
...
char *newmem;
...
newmem = (char *)sbrk(8192);
newmem[0] = 'Y';
```


4 malloc-Funktion

- malloc verwaltet einen vom Betriebssystem angeforderten Speicherbereich
 - welche Bereiche (Position, Länge) wurden vergeben
 - welche Bereiche sind frei
- Informationen über freie und belegte Speicherbereiche werden in Verwaltungsdatenstrukturen gehalten

```
struct mblock {  
    size_t size;  
    struct mblock *next;  
}
```

- Die Verwaltungsdatenstrukturen liegen jeweils vor dem zugehörigen Speicherbereich
- Die Verwaltungsdatenstrukturen der freien Speicherbereiche sind untereinander verkettet, bei vergebenen Speicherbereichen enthält next den Wert 0x00beef00

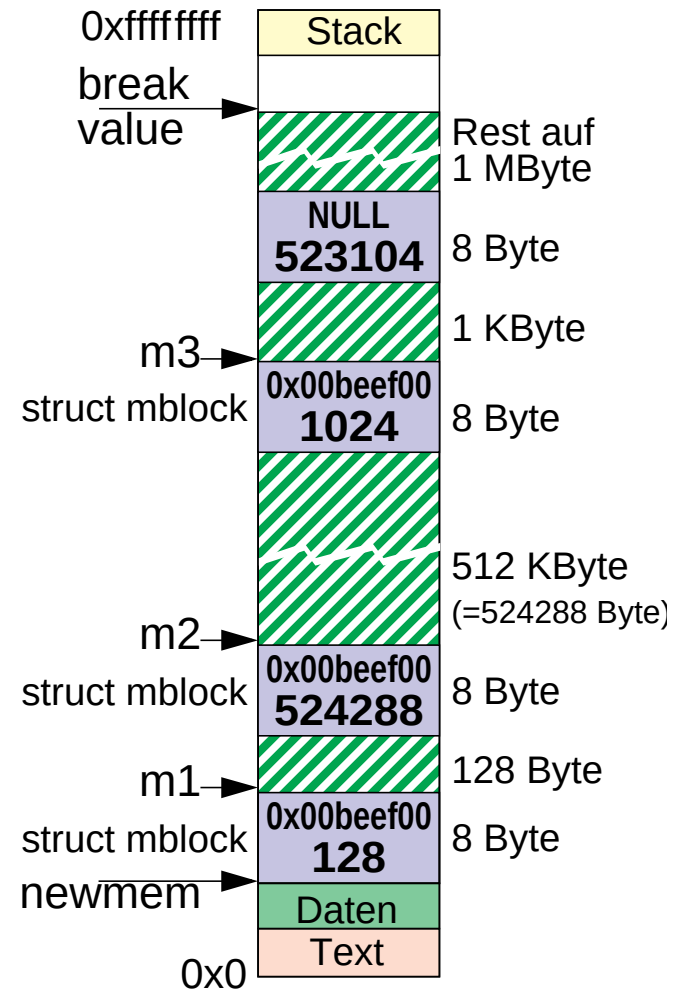
4 malloc-Funktion

■ Beispiel für die Situation nach 3 malloc-Aufrufen

```

...
char *m1, m2, m3;
...
m1 = (char *)malloc(128);
m2 = (char *)malloc(512*1024);
m3 = (char *)malloc(1024);

```

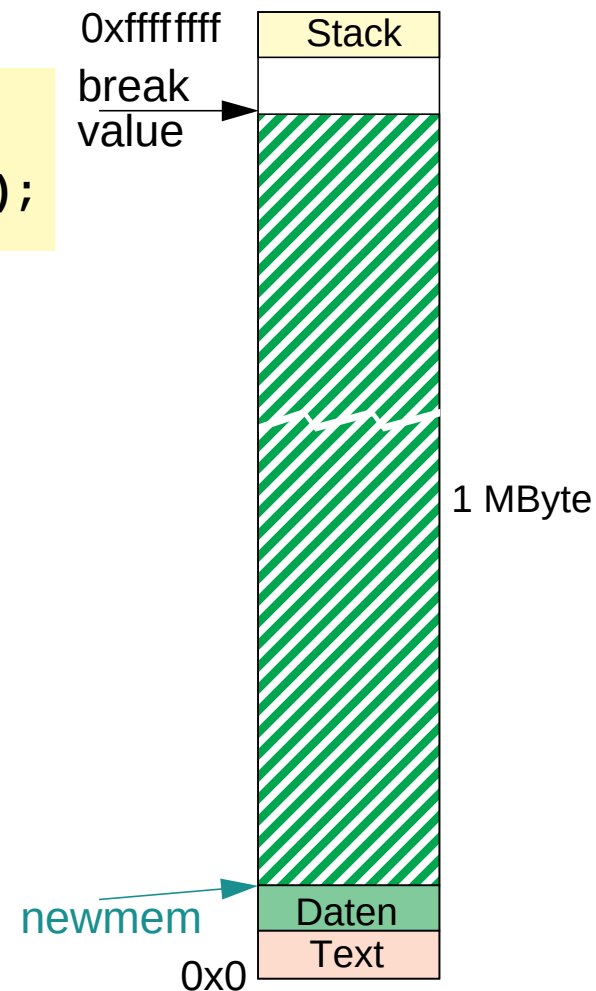


5 malloc-Interna - Initialisierung

■ initialer Zustand nach sbrk

◆ Speicher mit sbrk anfordern

```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```



5 malloc-Interns - Initialisierung (2)

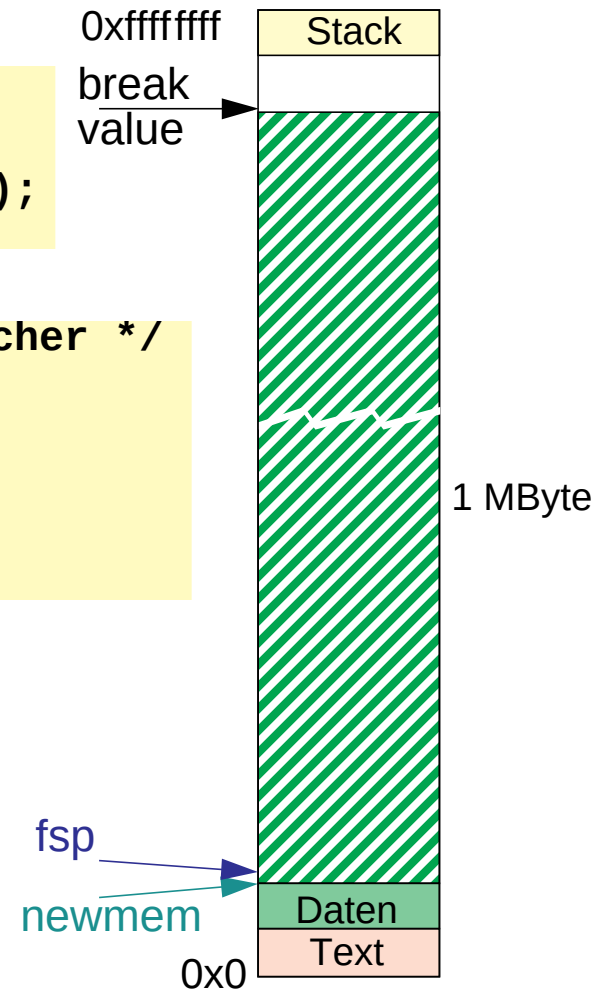
■ initialer Zustand nach sbrk

◆ Speicher mit sbrk anfordern

```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```

◆ struct mblock "hineinlegen"

```
struct mblock *fsp; /* Freispeicher */
...
fsp = (struct mblock *)newmem;
```



5 malloc-Internia - Initialisierung (3)

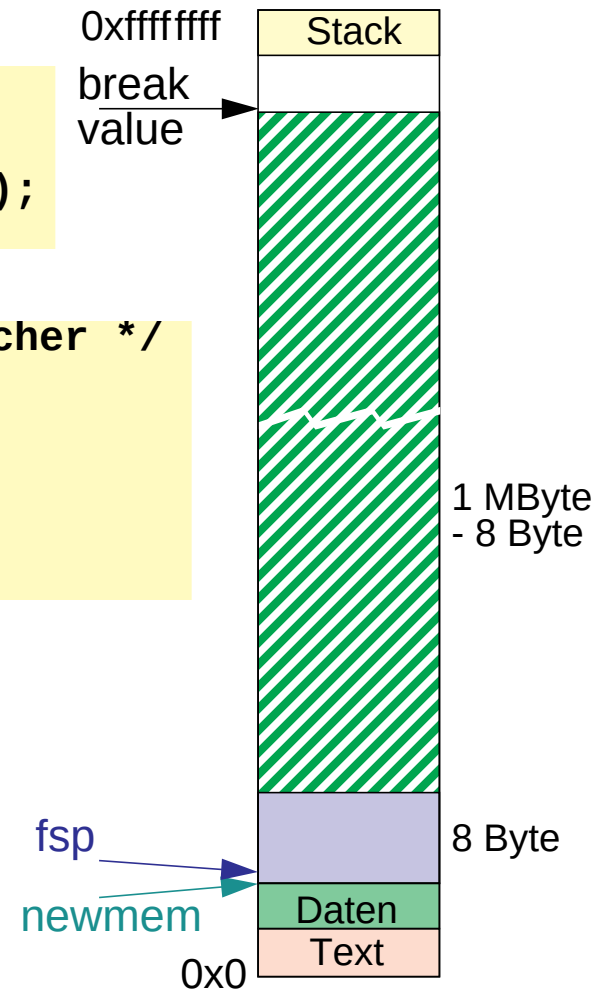
■ initialer Zustand nach sbrk

◆ Speicher mit sbrk anfordern

```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```

◆ struct mblock "hineinlegen"

```
struct mblock *fsp; /* Freispeicher */
...
fsp = (struct mblock *)newmem;
```



5 malloc-Interns - Initialisierung (4)

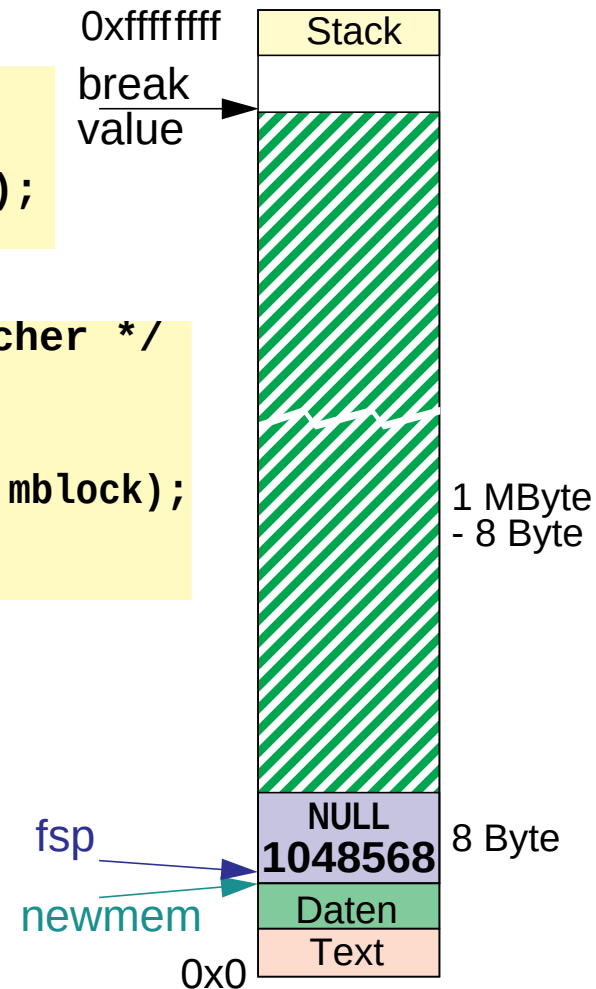
■ initialer Zustand nach sbrk

◆ Speicher mit sbrk anfordern

```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```

◆ struct mblock "hineinlegen"

```
struct mblock *fsp; /* Freispeicher */
...
fsp = (struct mblock *)newmem;
fsp->size = 1024*1024-sizeof(struct mblock);
fsp->next = NULL;
```



5 malloc-Interns - Initialisierung (5)

■ initialer Zustand nach sbrk

◆ Speicher mit sbrk anfordern

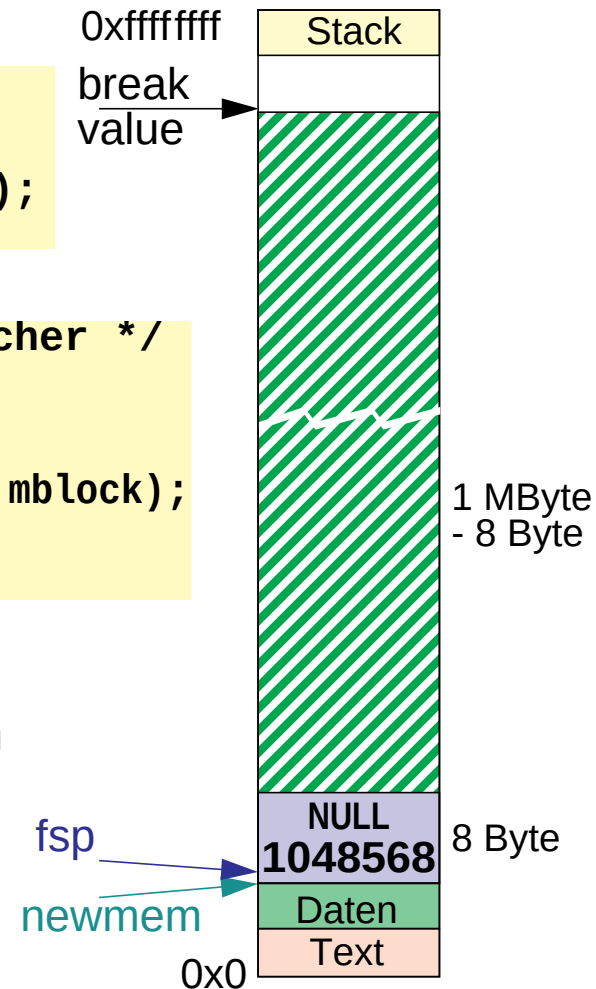
```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```

◆ struct mblock "hineinlegen"

```
struct mblock *fsp; /* Freispeicher */
...
fsp = (struct mblock *)newmem;
fsp->size = 1024*1024-sizeof(struct mblock);
fsp->next = NULL;
```

➔ zwei Zeiger mit unterschiedlichem Typ zeigen auf den gleichen Speicherbereich

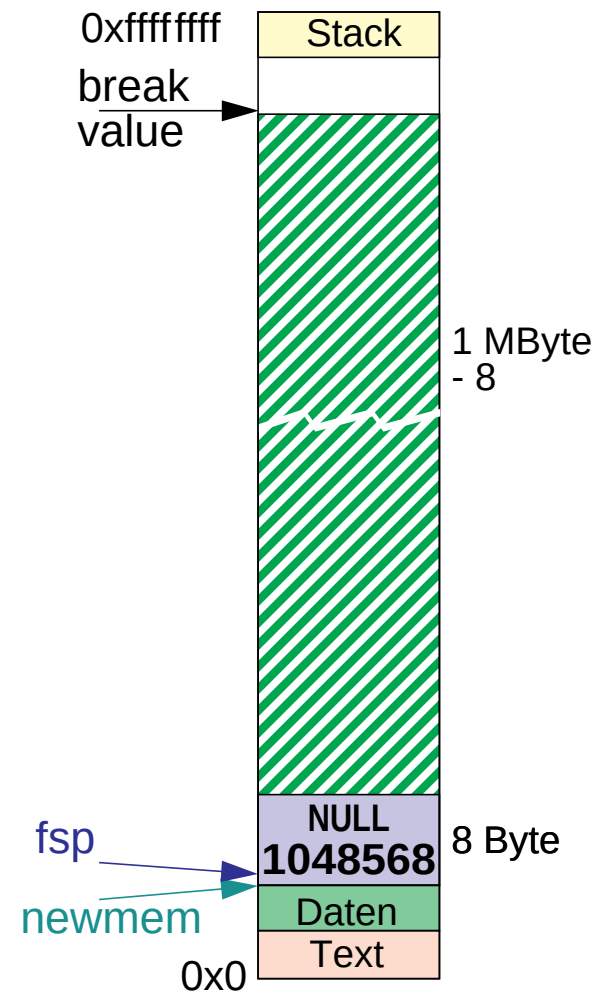
- unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponentenzugriffe)



6 malloc-Interna - Speichieranforderung

■ Aufgaben bei einer Speichieranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

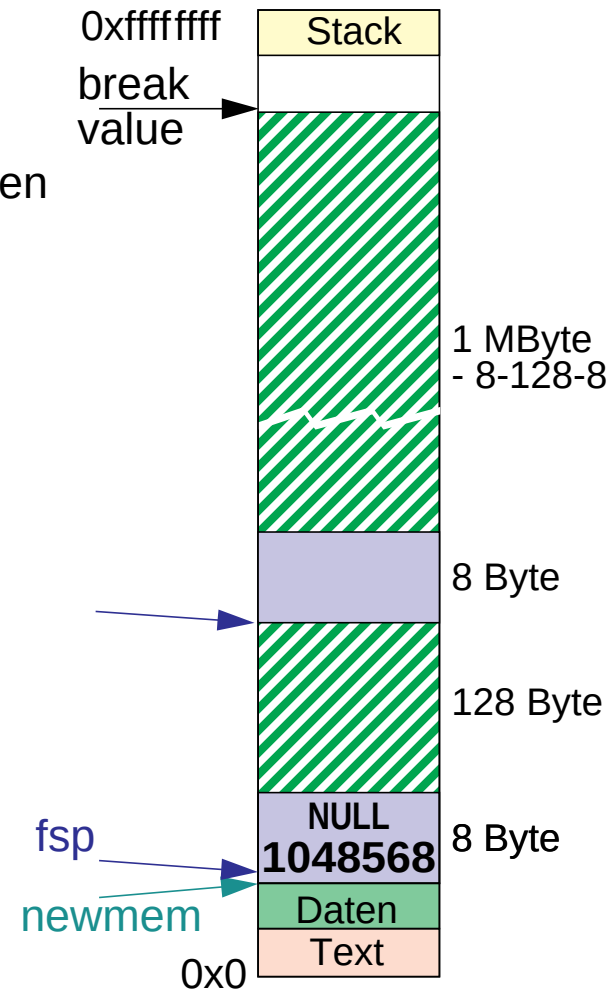


6 malloc-Interna - Speichieranforderung (2)

■ Aufgaben bei einer Speichieranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

- ◆ 128 Byte hinter dem fsp-mblock reservieren
- ◆ neuen mblock dahinter anlegen

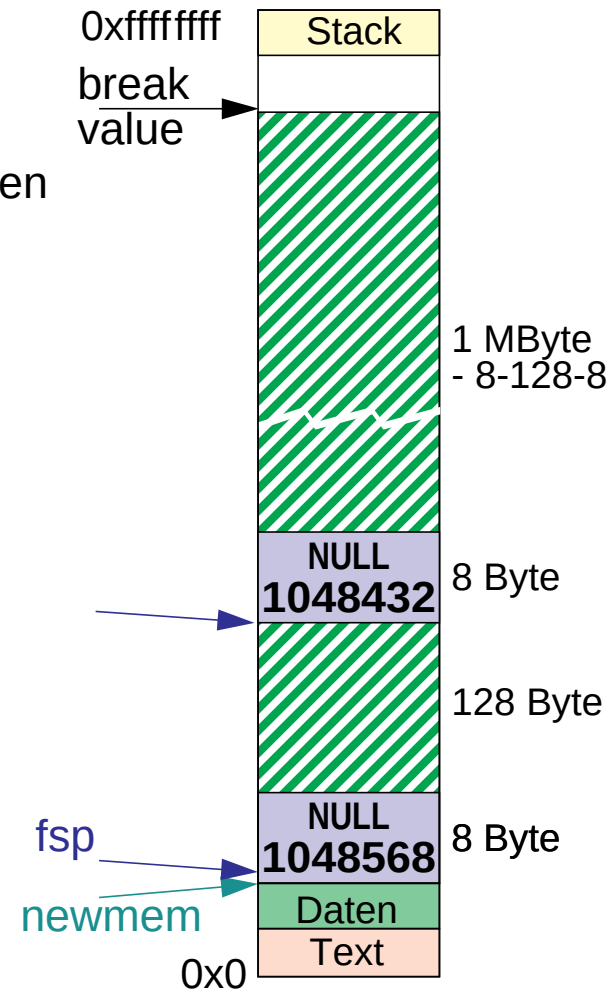


6 malloc-Interna - Speichieranforderung (3)

■ Aufgaben bei einer Speichieranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

- ◆ 128 Byte hinter dem fsp-mblock reservieren
- ◆ neuen mblock dahinter anlegen und initialisieren

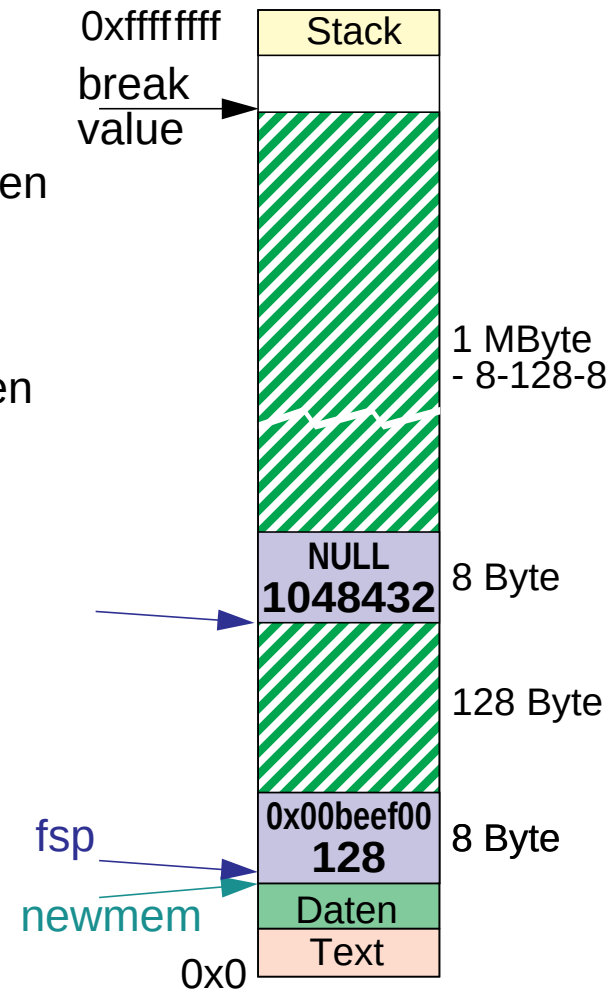


6 malloc-Interna - Speichieranforderung (4)

■ Aufgaben bei einer Speichieranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

- ◆ 128 Byte hinter dem fsp-mblock reservieren
- ◆ neuen mblock dahinter anlegen und initialisieren
- ◆ bisherigen fsp-mblock als belegt markieren

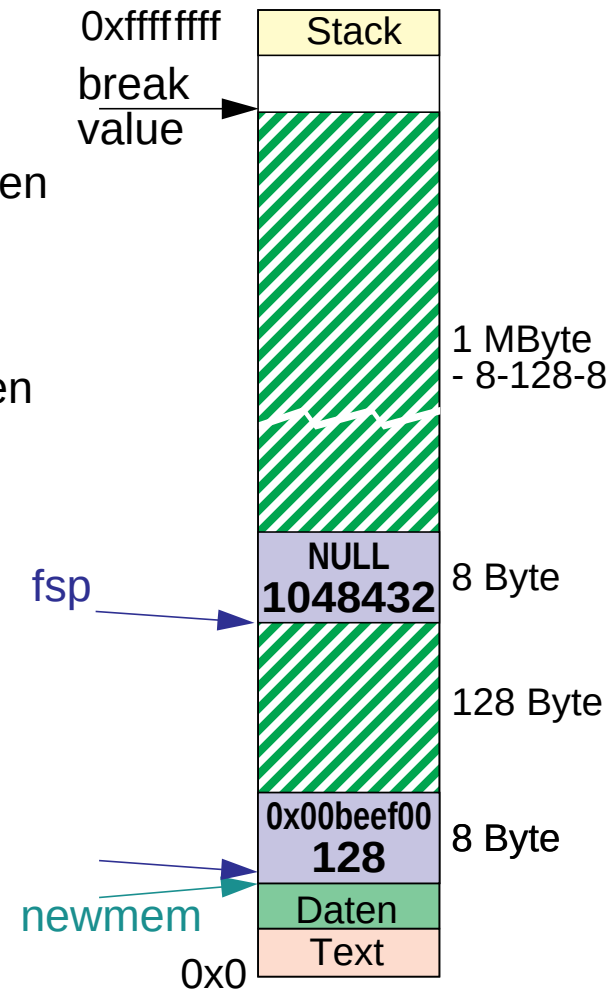


6 malloc-Interna - Speichieranforderung (5)

■ Aufgaben bei einer Speichieranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

- ◆ 128 Byte hinter dem fsp-mblock reservieren
- ◆ neuen mblock dahinter anlegen und initialisieren
- ◆ bisherigen fsp-mblock als belegt markieren
- ◆ fsp-Zeiger auf neuen mblock setzen

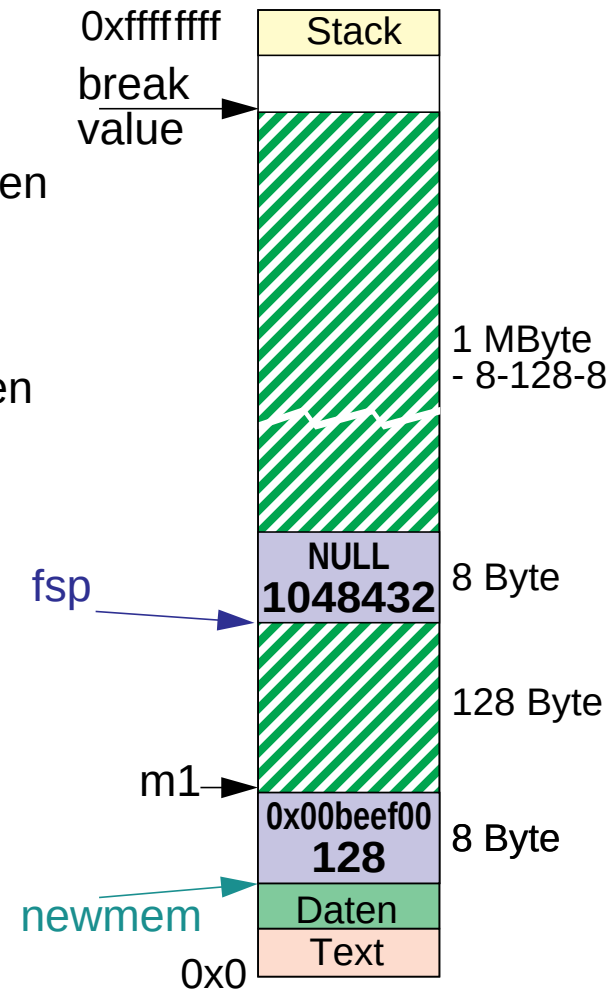


6 malloc-Interna - Speichieranforderung (6)

■ Aufgaben bei einer Speichieranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

- ◆ 128 Byte hinter dem fsp-mblock reservieren
- ◆ neuen mblock dahinter anlegen und initialisieren
- ◆ bisherigen fsp-mblock als belegt markieren
- ◆ fsp-Zeiger auf neuen mblock setzen
- ◆ Zeiger auf die reservierten 128 Byte zurückgeben



6 malloc-Interna - Speichieranforderung (7)

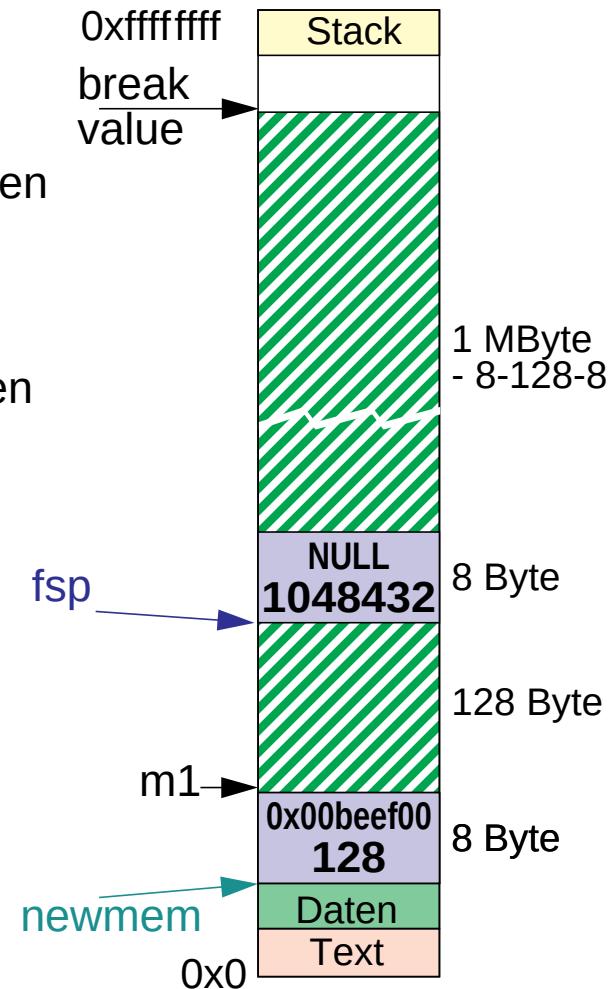
■ Aufgaben bei einer Speichieranforderung

```
char *m1;
m1 = (char *)malloc(128);
```

- ◆ 128 Byte hinter dem fsp-mblock reservieren
- ◆ neuen mblock dahinter anlegen und initialisieren
- ◆ bisherigen fsp-mblock als belegt markieren
- ◆ fsp-Zeiger auf neuen mblock setzen
- ◆ Zeiger auf die reservierten 128 Byte zurückgeben

■ Frage: wie rechnet man auf dem Speicher?

- in `char *` ?
- in `struct mblock *` ?



7 malloc-Interna - Zeigerarithmetik

- Problem: Verwaltungsdatenstrukturen sind mblock-Strukturen, angeforderte Datenbereiche sind Byte-Felder
 - Zeigerarithmetik muss teilweise mit struct mblock-Einheiten, teilweise mit char-Einheiten operieren
- Variante 1: Berechnungen von fsp_neu in Byte-/char-Einheiten

```
void *malloc(size_t size) {  
    struct mblock *fsp_neu, *fsp_alt;  
    fsp_alt = fsp;  
    ...  
    fsp_neu = (struct mblock *) ((char *)fsp_alt  
                                + sizeof(struct mblock) + size);  
    ...  
    return((void *) (fsp_alt + 1));  
}
```

7 malloc-Internia - Zeigerarithmetik (2)

■ Variante 2: Berechnungen in struct mblock-Einheiten

```
void *malloc(size_t size) {
    struct mblock *fsp_neu, *fsp_alt;
    int units;
    fsp_alt = fsp;
    ...
    units = ( (size-1) / sizeof(struct mblock) ) + 1;
    fsp_neu = fsp + 1 + units;
    ...
    return((void *)(fsp_alt + 1));
}
```

- ◆ Unterschied: bei der Umrechnung von size auf units wird auf die nächste ganze struct mblock-Einheit aufgerundet
- ◆ Vorteil: die mblock-Strukturen liegen nach einer Anforderung von "krummen" Speichermengen nicht auf "ungeraden" Speichergrenzen
 - manche Prozessoren fordern, dass int-Werte immer auf Wortgrenzen (durch 4 teilbar) liegen (sonst Trap: Bus error beim Speicherzugriff)
 - bei Intel-Prozessoren: ungerade Positionen zwar erlaubt, aber ineffizient
 - aber: veränderte Größe in den Verwaltungsstrukturen beachten!

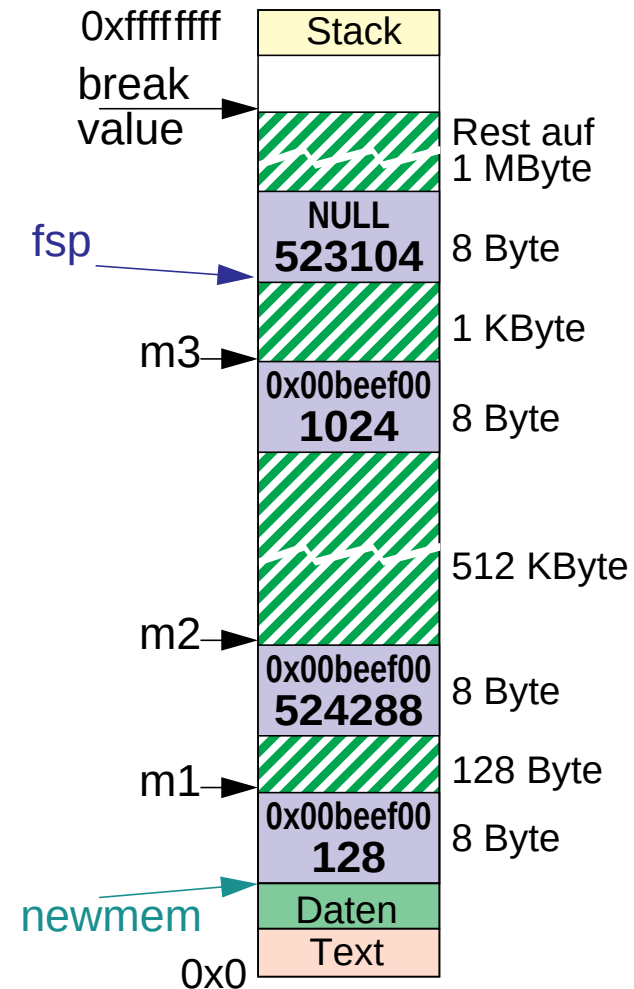
8 malloc-Interna - Speicher freigeben

■ Situation nach 3 malloc-Aufrufen

```

...
char *m1, *m2, *m3;
...
m1 = (char *)malloc(128);
m2 = (char *)malloc(512*1024);
m3 = (char *)malloc(1024);

```



8 malloc-Interna - Speicher freigeben (2)

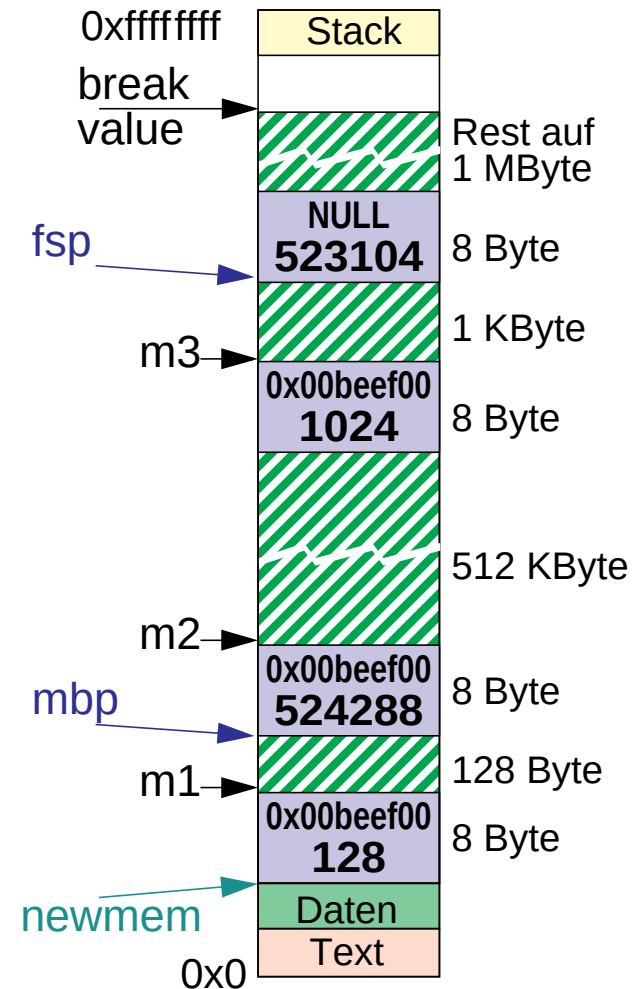
■ Freigabe von m2 - Aufgaben

```

...
char *m1, *m2, *m3;
...
m1 = (char *)malloc(128);
m2 = (char *)malloc(512*1024);
m3 = (char *)malloc(1024);
...
free(m2);

```

- ◆ Zeiger **mbp** auf zugehörigen mblock ermitteln
- ◆ überprüfen, ob ein gültiger, belegter mblock vorliegt (0x00beef00!)



8 malloc-Interna - Speicher freigeben (3)

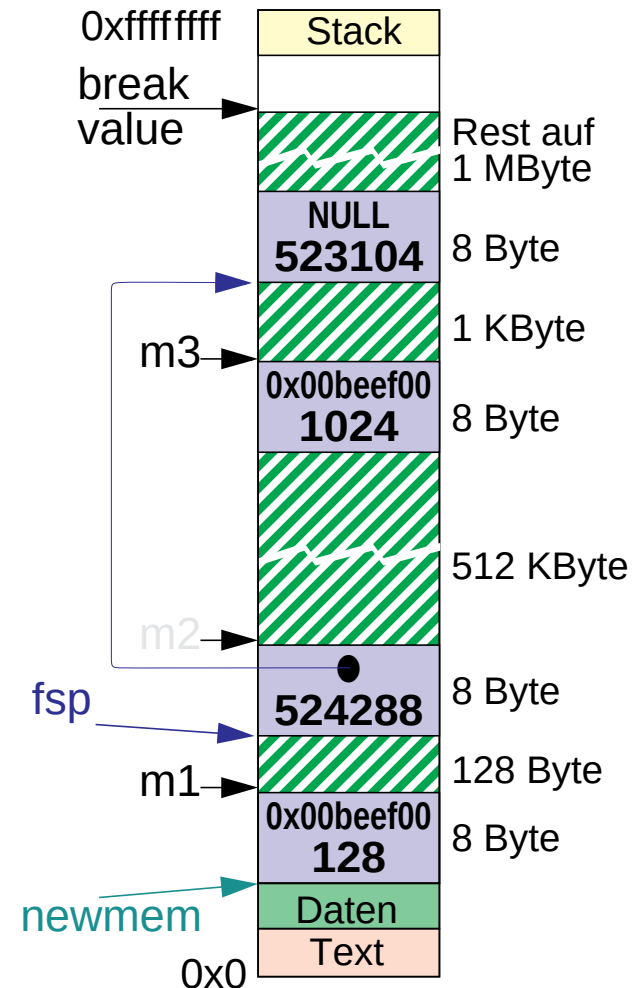
■ Freigabe von m2 - Aufgaben

```

...
char *m1, *m2, *m3;
...
m1 = (char *)malloc(128);
m2 = (char *)malloc(512*1024);
m3 = (char *)malloc(1024);
...
free(m2);

```

- ◆ Zeiger **mbp** auf zugehörigen mblock ermitteln
- ◆ überprüfen, ob ein gültiger, belegter mblock vorliegt (0x00beef00!)
- ◆ **fsp** auf freigegebenen Block setzen, bisherigen fsp-mblock verketten

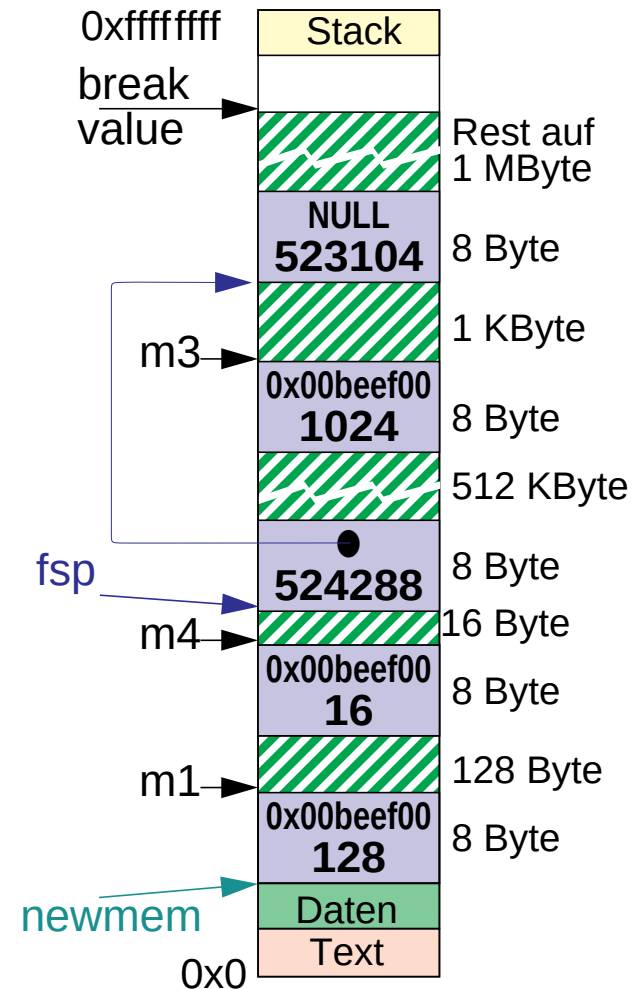


9 malloc-Internia - erneut Speicher anfordern

- neue Anforderung von 10 Byte

```
...
char *m4
...
m4 = (char *)malloc(10);
```

- ◆ Annahme: Zeigerberechnung in struct mblock-Einheiten (mit Aufrunden)
- ◆ neuen mblock danach anlegen
- ◆ überprüfen, ob ein gültiger, belegter mblock vorliegt (0x00beef00!)
- ◆ `fsp` auf freigegebenen Block setzen, bisherigen `fsp`-mblock verketteten



10 malloc - abschließende Bemerkungen

- sehr einfache Implementierung - in der Praxis problematisch
 - ◆ Speicher wird im Laufe der Zeit stark fragmentiert
 - Suche nach passender Lücke dauert zunehmend länger
 - es kann passieren, dass keine passende Lücke mehr zu finden ist, obwohl insgesamt genug Speicher frei wäre
 - Verschmelzung von benachbarten freigegebenen Bereichen wäre notwendig
- sinnvolle Implementierung erfordert geeignete Speichervergabestrategie
 - Implementierung erheblich aufwändiger - Resultat aber entsprechend effizienter
 - Strategien werden im Abschnitt Speicherverwaltung in der Vorlesung behandelt
(z. B. best fit, worst fit oder Buddy-Verfahren)