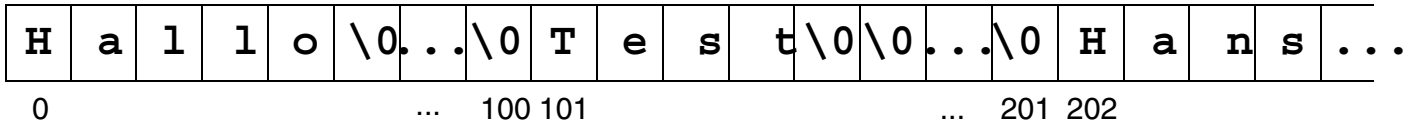
U4 4. Übung

- Aufgabe 2: qsort - Fortsetzung
- Dynamische Speicherallokation
- Fehlerbehandlung Reloaded
- Infos zur Aufgabe 4: malloc-Implementierung

U4-1 Aufgabe 2: Sortieren mittels qsort

1 wsort - Datenstrukturen (1. Möglichkeit)

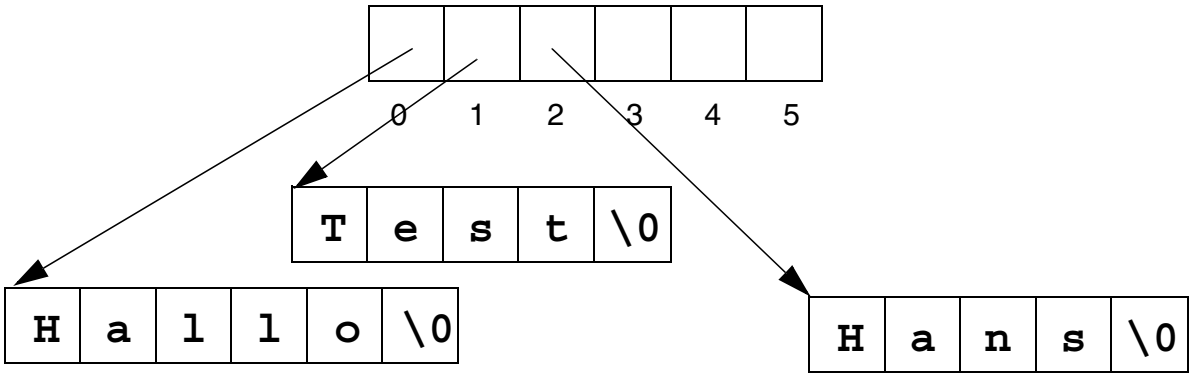
- Array von Zeichenketten



- Vorteile:
  - einfach
- Nachteile:
  - hoher Kopieraufwand (303 Bytes pro Umordnung)
  - Maximale Länge der Worte muss bekannt sein
  - Verschwendung von Speicherplatz

2 wsort - Datenstrukturen (2. Möglichkeit)

- Array von Zeigern auf Zeichenketten



- Vorteile:
  - schnell, da nur Zeiger vertauscht werden (x86-32: 12 Byte pro Umordnung)
  - 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(*((const char **)a), *((const char **)b));
}
```

  - beim qsort-Aufruf:

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

U4-2 Verwendung dynamischer Speicherallokation

- Beispiel (aus Abgaben zu Aufgabe 2):

```
char *buffer = (char *) malloc(102 * sizeof(char));
if ( NULL == buffer ) {
    perror("malloc"); exit(EXIT_FAILURE);
}
while (fgets(buffer, 102, stdin) != NULL) {
    ... strcpy(somewhere_else, buffer); ...
}
free(buffer);
```

- teure Allokations- und Freigabeoperationen (siehe Aufgabe 4)
- erfordert Fehlerbehandlung
- viel Schreibarbeit
  - verschlechtert Code-Lesbarkeit
  - kostet Zeit wenn keine da ist (z.B. in der Klausur)

U4-2 Verwendung dynamischer Speicherallokation

- Alternative: Stackallokation

```
char buffer[102];

while (fgets(buffer, 102, stdin) != NULL) {
    ... strcpy(somewhere_else, buffer); ...
}
```

- Sehr effizient
  - Allokation: Stackpointer -= 102;
  - Freigabe: Stackpointer += 102;
- Keine Fehlerbehandlung durch das Programm
  - Stacküberlauf wird ggf. vom Betriebssystem erkannt (`SIGSEGV`)
- Implizite Freigabe beim Verlassen der Funktion
  - keine Speicherlecks möglich

U4-3 Fehlerbehandlung Reloaded

- Fehlermeldungen und Warnungen immer auf den Standardfehlerkanal
  - auch Warnungen des Programms: z.B. "Wort zu lang"
- Keine ungeforderten Ausgaben auf die Standardausgabe
  - wie z.B. "Hier kommt die sortierte Ausgabe"
  - beeinträchtigt die Verwendbarkeit des Programms in der Stapelverarbeitung
  - erschwert die Vergleichbarkeit mit anderen Lösungen

U4-3 Fehlerbehandlung Reloaded

- Signalisierung von Fehlern normalerweise durch Rückgabewert
- Nicht bei allen Funktionen möglich, z.B. **fgets(3)**

```
while (fgets(buffer, 102, stdin) != NULL) {
    ...
}

/* EOF oder Fehler? */
```

- Rückgabewert `NULL` sowohl im Fehlerfalls als auch bei `End-of-File`

U4-3 Fehlerbehandlung Reloaded

- Erkennung im Fall von I/O-Streams mit **ferror(3)** und **feof(3)**

```
while (fgets(buffer, 102, stdin) != NULL) {
    ...
}

/* EOF oder Fehler? */
if(ferror(stdin)) {
    /* Fehler */
    ...
}
```

U4-3 Fehlerbehandlung Reloaded

- Nicht in allen Fällen existieren solche Spezialfunktionen
- Allgemeiner Ansatz durch Setzen und Prüfen von `errno`

```
#include <errno.h>

while (errno=0, fgets(buffer, 102, stdin) != NULL) {
    ... /* keine break-Statements in der Schleife */
}
/* EOF oder Fehler? */
if(errno != 0) {
    /* Fehler */
    ...
}
```

- `errno=0` *unmittelbar* vor Aufruf der problematischen Funktion
  ➥ `errno` wird nur im Fehlerfall gesetzt und bleibt sonst evtl. unverändert
- Abfrage der `errno` *unmittelbar* nach Rückgabe des pot. Fehlerwerts
  ➥ `errno` könnte sonst durch andere Funktion verändert werden

U4-4 Aufgabe 4: einfache malloc-Implementierung

1 Überblick

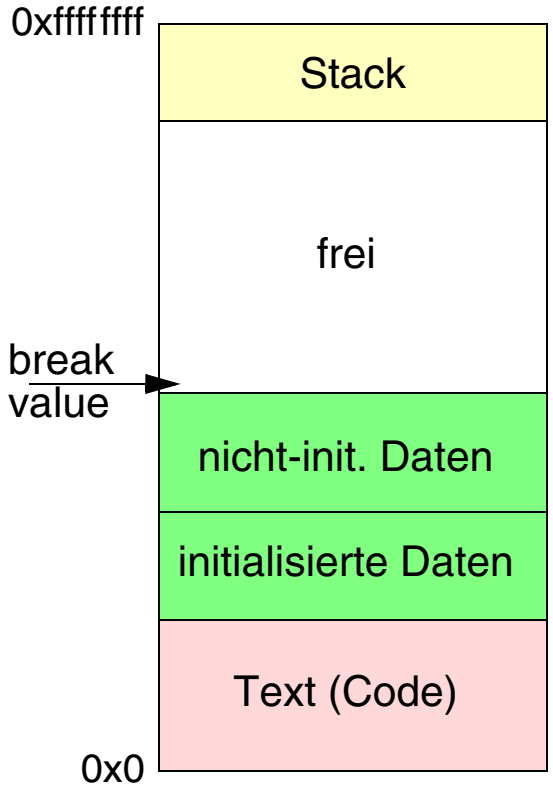
- erheblich vereinfachte Implementierung
  - ➤ nur einmal am Anfang Speicher vom Betriebssystem anfordern (1 MB)
  - ➤ First-Fit-Allokationsstrategie
  - ➤ freigegebener Speicher wird in einer einfachen verketteten Liste verwaltet (benachbarte freie Blöcke werden nicht mehr verschmolzen)
  - ➤ `realloc` 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

- Größe des Datensegments ändern



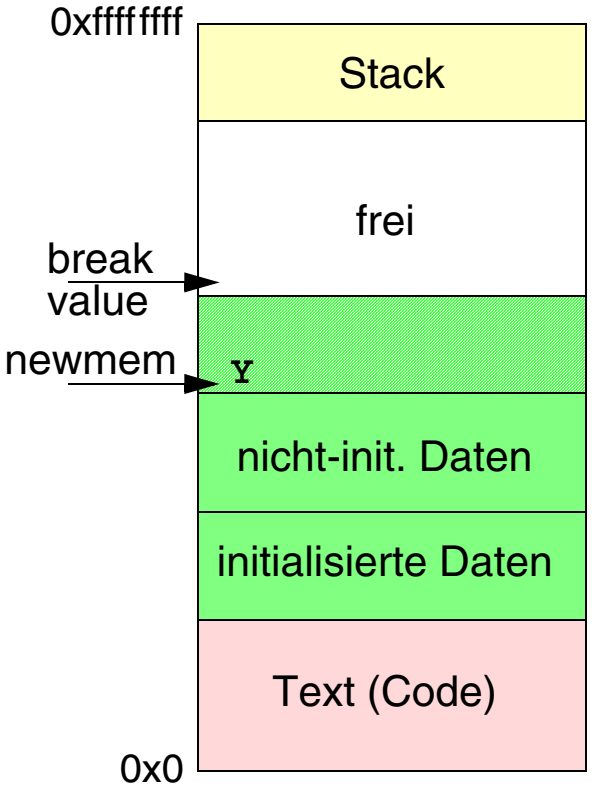
- break value: Ende des Datensegments
- **brk(2)** erlaubt es, diese Adresse neu festzulegen
  - ➤ zusätzlicher Speicher hinter den nicht-initialisierten Daten
- Schnittstellen:

```
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)

- Größe des Datensegments ändern



- 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

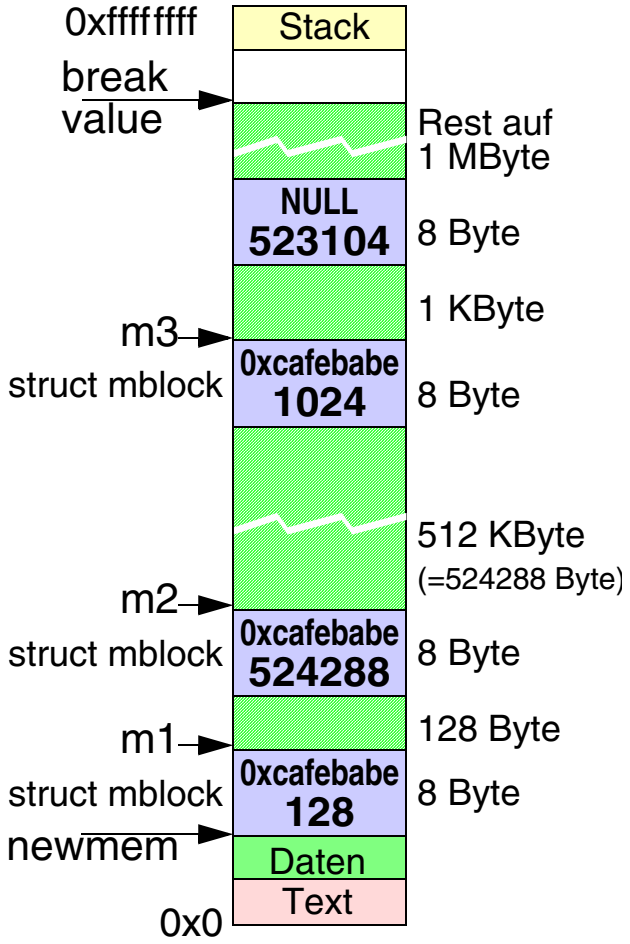
```
struct mblock {
    size_t size;
    struct mblock *next;
}
```

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

4 malloc-Funktion

- Beispiel für die Situation nach 3 malloc-Aufrufen (32-Bit-Architektur!)

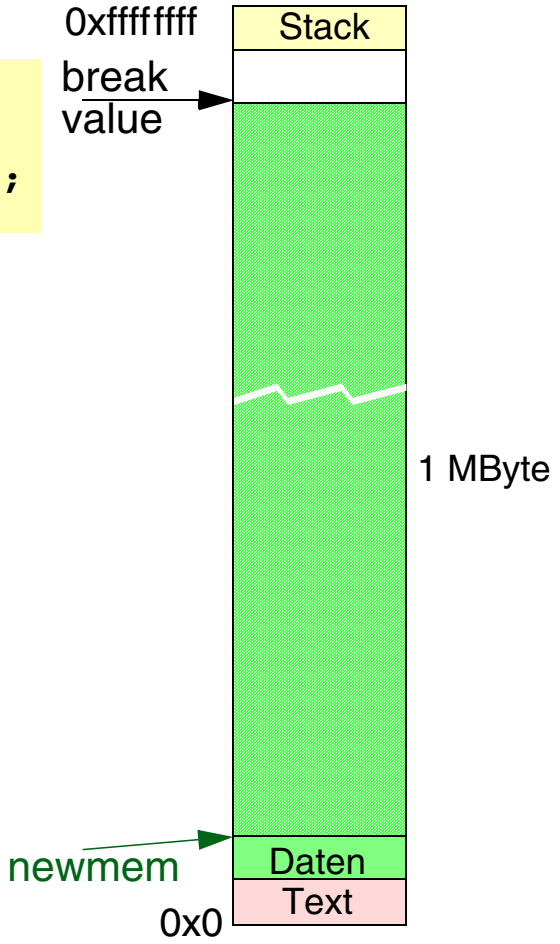
```
...
char *m1, *m2, *m3;
...
m1 = (char *)malloc(128);
m2 = (char *)malloc(512*1024);
m3 = (char *)malloc(1024);
```



5 malloc-Interna - Initialisierung

- initialer Zustand nach sbrk
  - Speicher mit sbrk anfordern

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



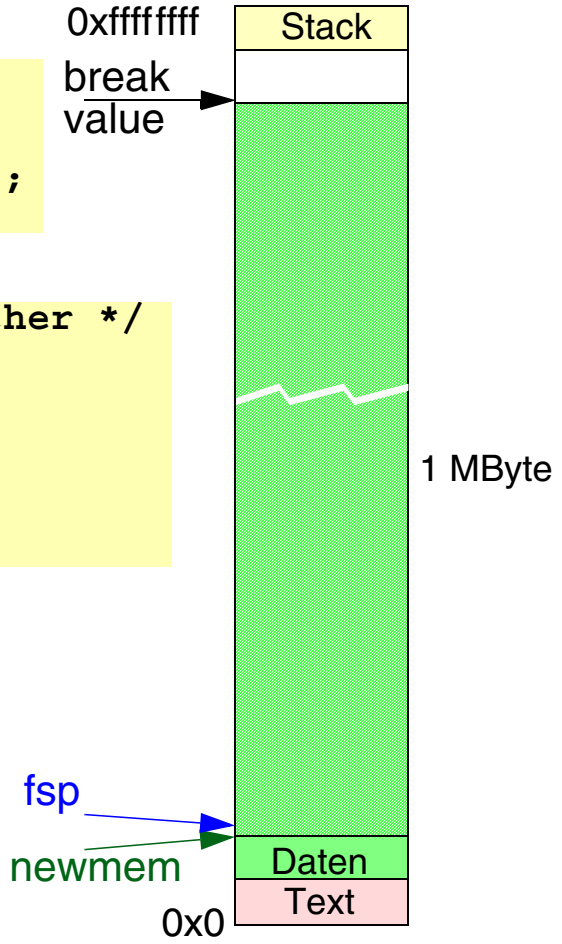
5 malloc-Interna - Initialisierung (2)

- 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;
```



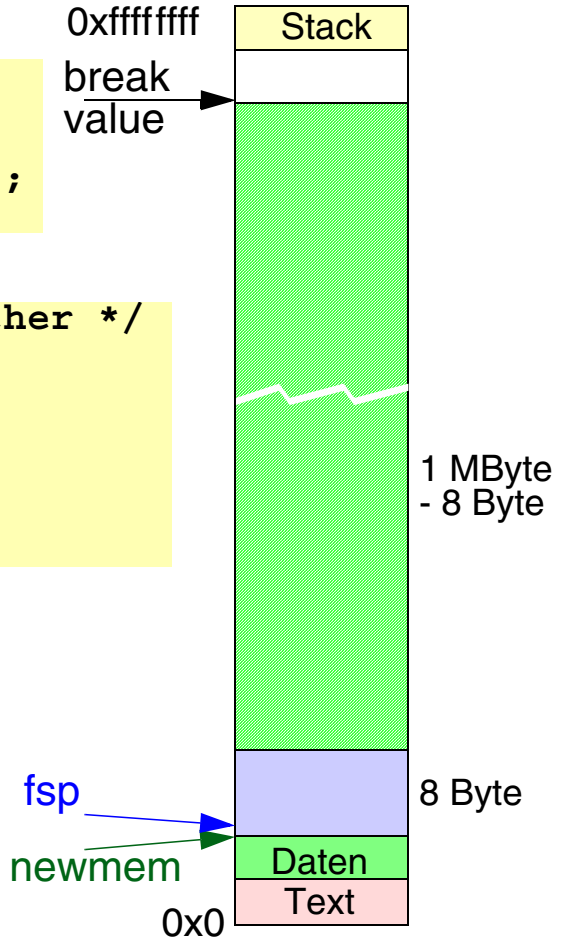
5 malloc-Interna - Initialisierung (3)

- 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;
```



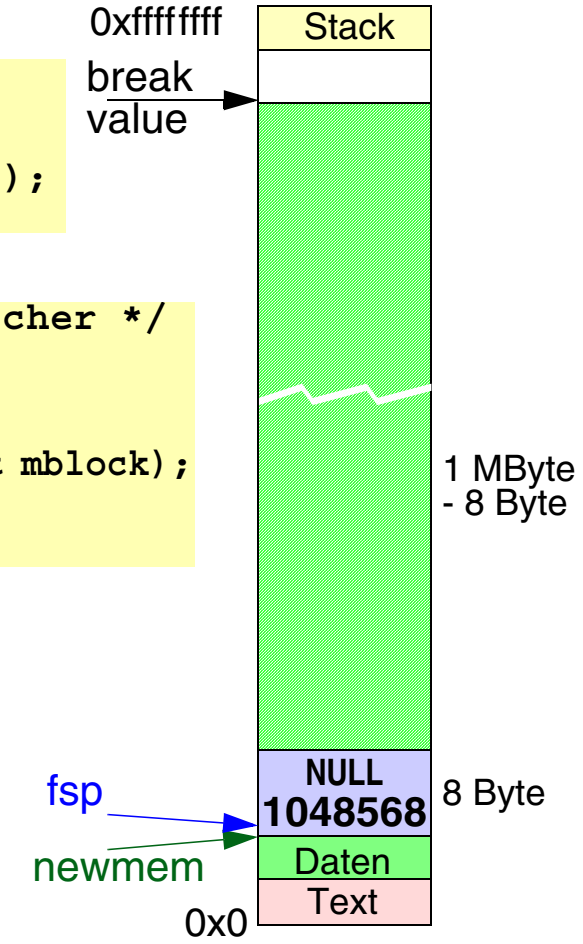
5 malloc-Interna - Initialisierung (4)

- 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-Interna - Initialisierung (5)

- initialer Zustand nach sbrk
  - Speicher mit sbrk anfordern

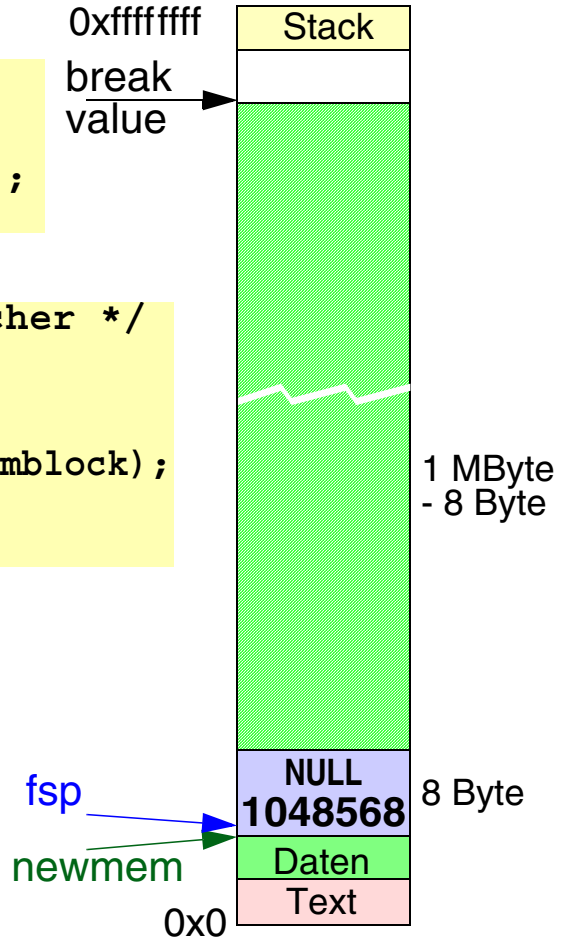
```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```

  - 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

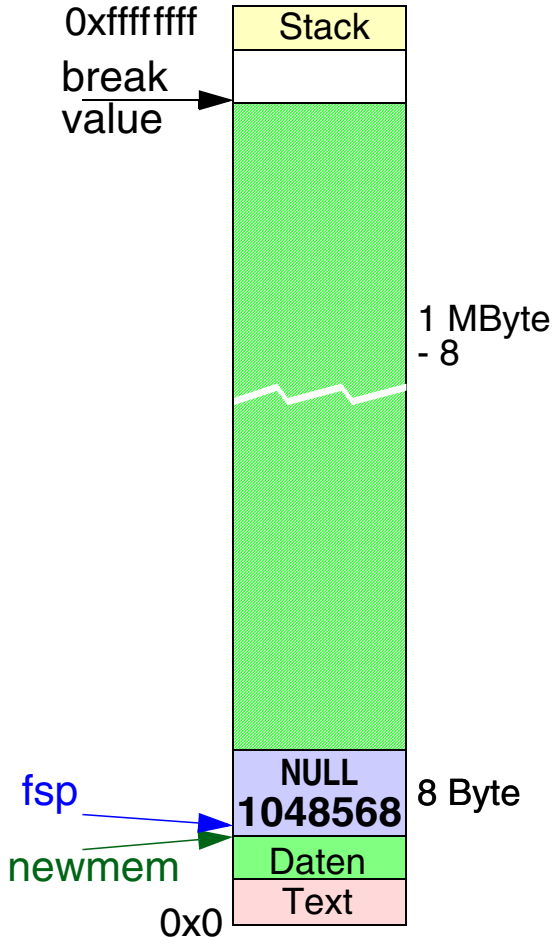
➤ unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponentenzugriffe)



6 malloc-Interna - Speicheranforderung

- Aufgaben bei einer Speicheranforderung

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

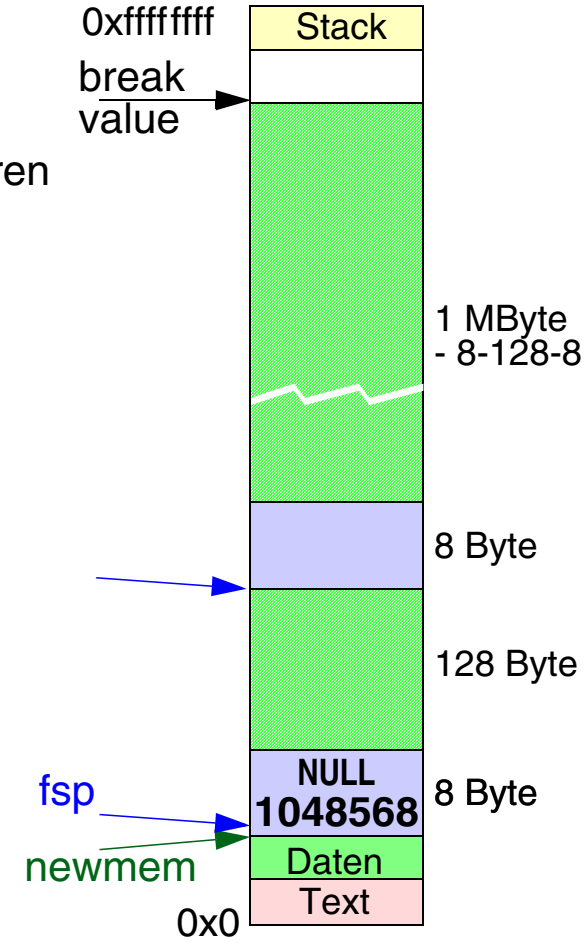


6 malloc-Interna - Speicheranforderung (2)

- Aufgaben bei einer Speicheranforderung

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

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

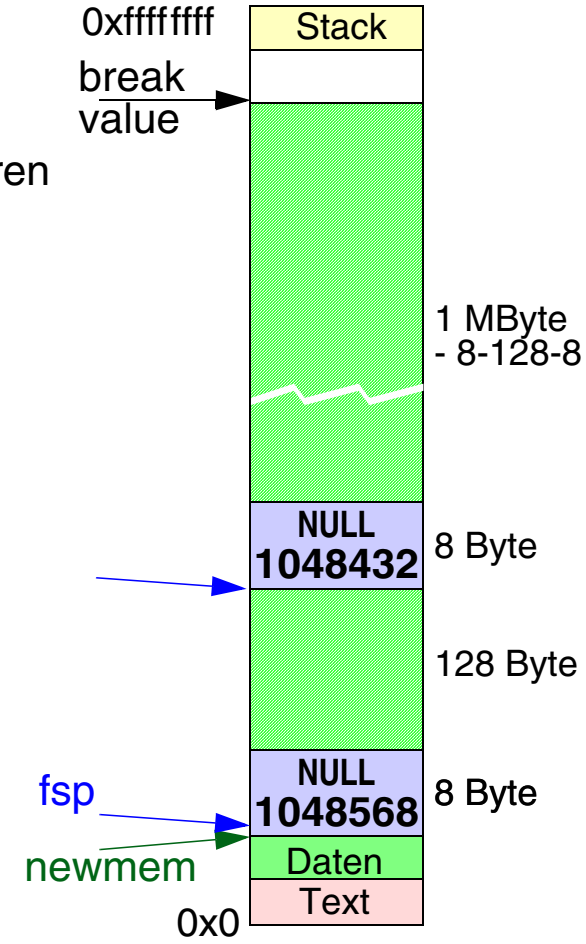


6 malloc-Interna - Speicheranforderung (3)

- Aufgaben bei einer Speicheranforderung

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

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

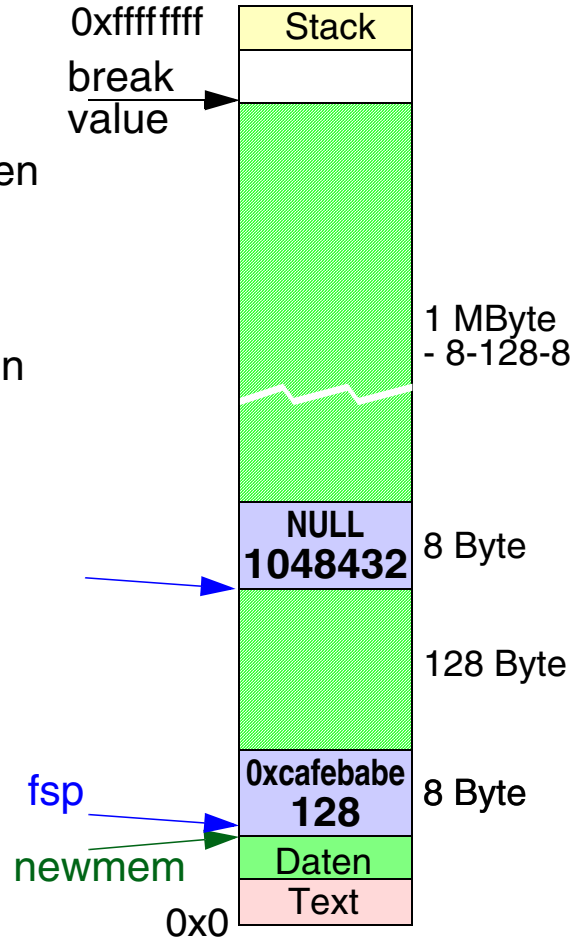


6 malloc-Interna - Speicheranforderung (4)

- Aufgaben bei einer Speicheranforderung

```
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

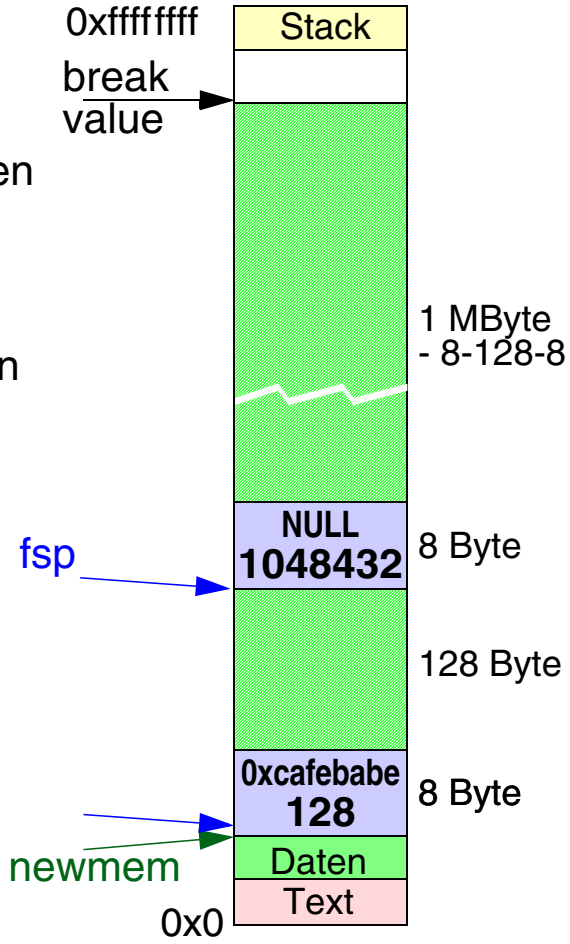


6 malloc-Interna - Speicheranforderung (5)

- Aufgaben bei einer Speicheranforderung

```
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

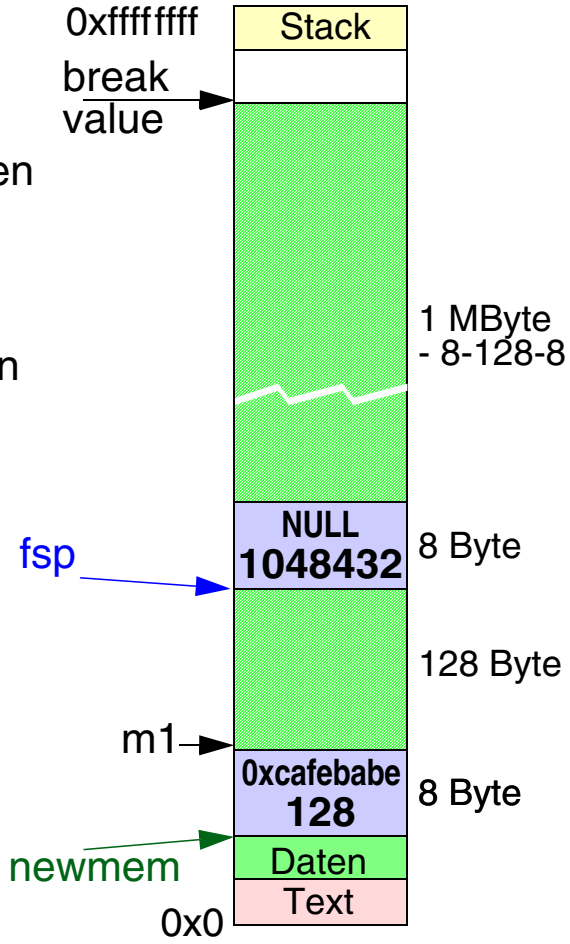


6 malloc-Interna - Speicheranforderung (6)

- Aufgaben bei einer Speicheranforderung

```
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



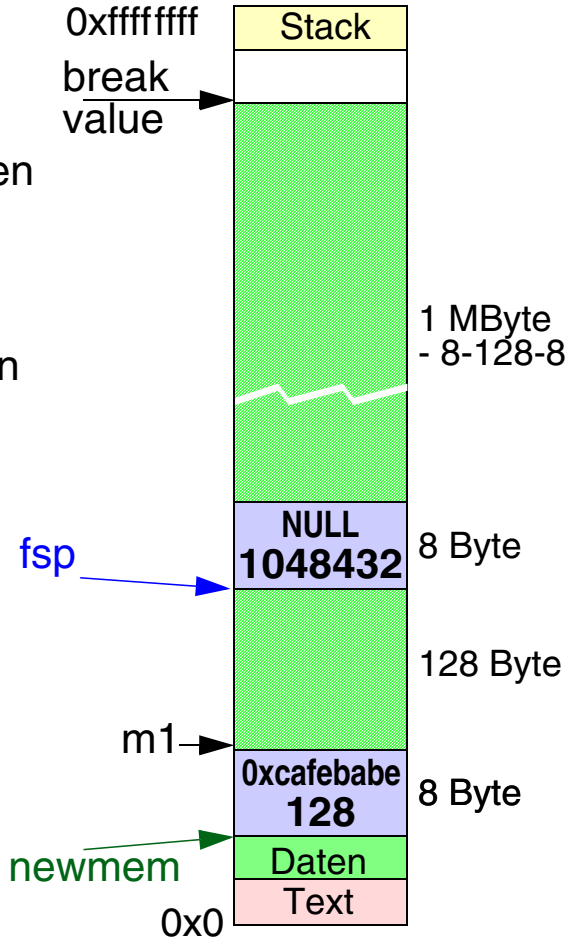
6 malloc-Interna - Speicheranforderung (7)

- Aufgaben bei einer Speicheranforderung

```
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-Interna - Zeigerarithmetik (2)

- Variante 2: Berechnungen in struct mblock-Einheiten

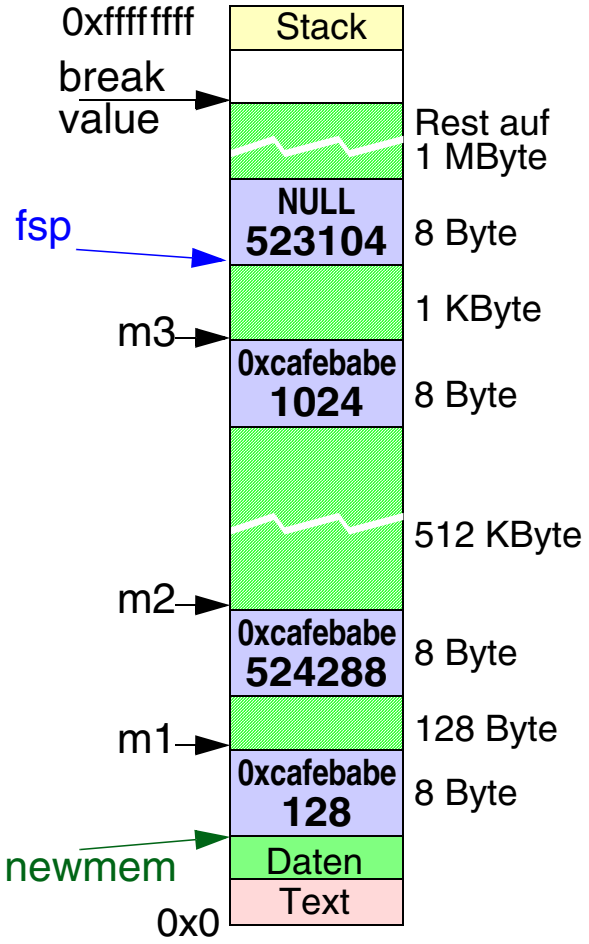
```
void *malloc(size_t size) {
    struct mblock *fsp_neu, *fsp_alt;
    int units;
    fsp_alt = fsp;
    ...
    units = ( (size-1) / sizeof(struct mblock) ) + 1;
    fsp_neu = fsp + 1 + units;
    ...
    return((void *)(fsp_alt + 1));
```

  - 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);
```

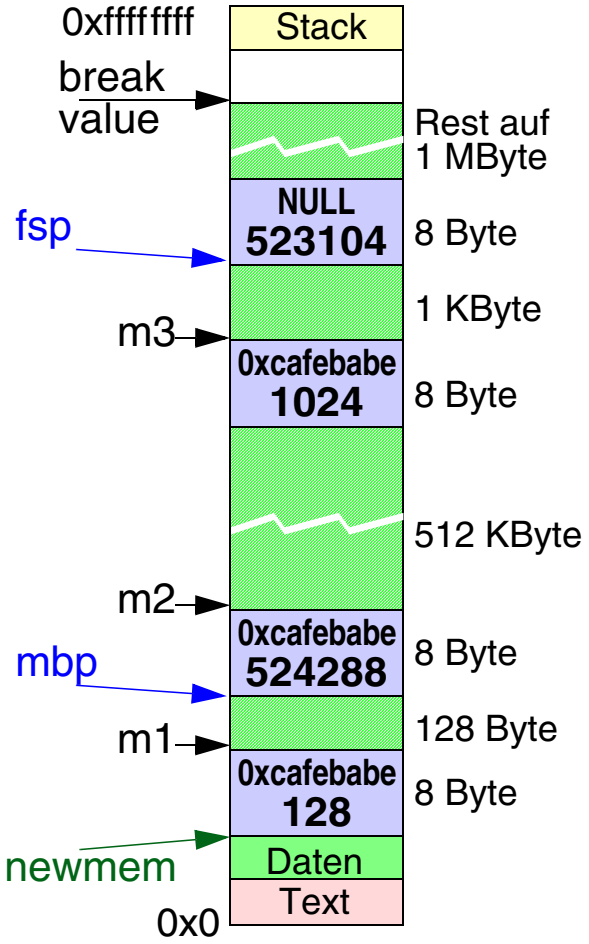


8 malloc-Interna - Speicher freigeben (2)

- 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 (0xcafebabe!)

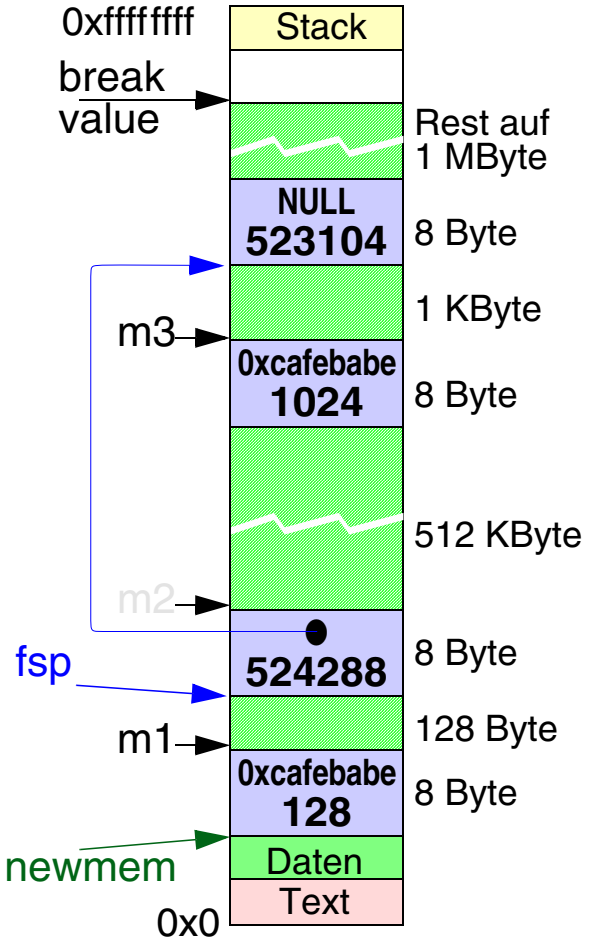


8 malloc-Interna - Speicher freigeben (3)

- 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 (0xcafebabe!)
- `fsp` auf freigegebenen Block setzen, bisherigen fsp-mblock verketten

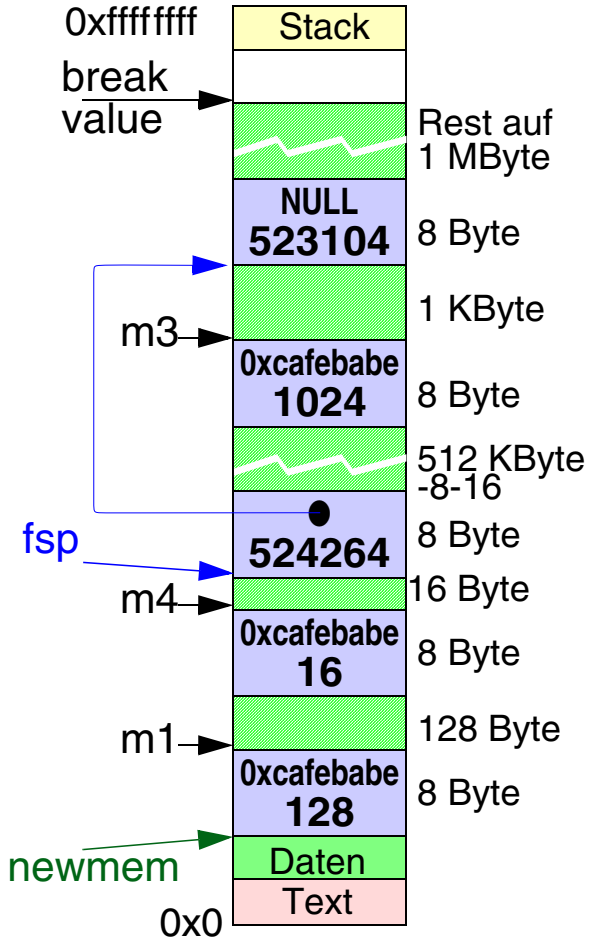


9 malloc-Interna - erneut Speicher anfordern

- neue Anforderung von 10 Byte

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

  - Annahme: Zeigerberechnung in struct mblock-Einheiten (mit Aufrunden => 16 Byte)
  - neuen mblock danach anlegen



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
    - ➤ evtl. keine passende Lücke mehr zu finden, obwohl insgesamt genug Speicher frei
    - ➤ Lösung: Verschmelzung benachbarter freigegebener Blöcke
- 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)