

U4 3. Übung

■ Aufgabe 3: malloc-Implementierung

U4-1 Aufgabe 3: einfache malloc-Implementierung

■ Ziele der Aufgabe

- ◆ Zusammenhang zwischen "nacktem Speicher" und typisierten Datenbereichen verstehen
- ◆ Funktion aus der C-Bibliothek selbst realisieren

■ Vereinfachungen:

- ◆ First-Fit-ähnliche Allokationsstrategie
- ◆ 1 MiB Speicher statisch allokiert
- ◆ freier Speicher wird in einer einfach-verketteten Liste (unsortiert) verwaltet
- ◆ benachbarte freie Blöcke werden nicht verschmolzen
- ◆ **realloc** verlängert den Speicher nicht, sondern wird **grundsätzlich** auf ein neues **malloc**, **memcpy** und **free** abgebildet

1 malloc-Funktion

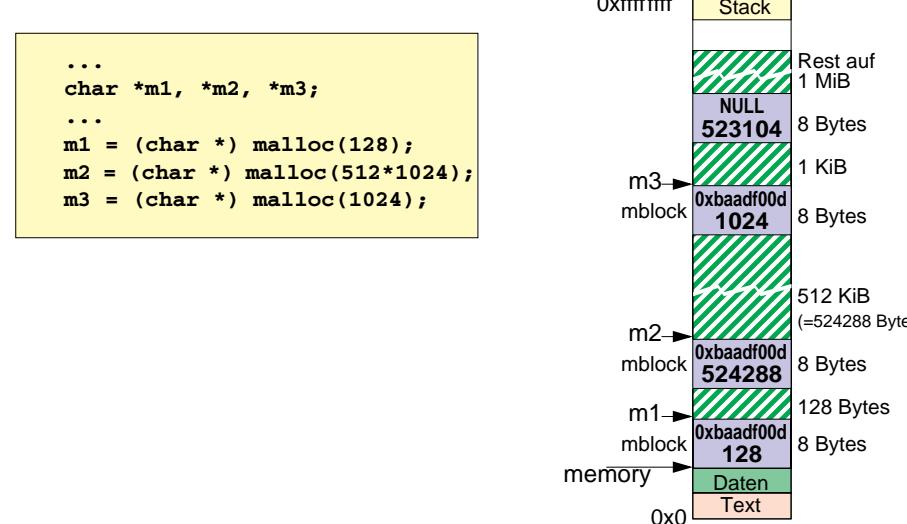
- verwaltet folgende Informationen über einen Speicherbereich
 - ◆ welche Bereiche wurden vergeben
 - ◆ welche Bereiche sind frei
- Informationen über freie und belegte Speicherbereiche werden in Verwaltungsdatenstrukturen gehalten

```
typedef struct mblock {
    size_t size;           // Größe des anhängenden Bereichs
    struct mblock *next;  // Verkettung freier Bereiche
} mblock;
```

- ◆ Die Verwaltungsdatenstrukturen liegen jeweils vor dem zugehörigen Speicherbereich
- Die Verwaltungsdatenstrukturen der freien Speicherbereiche sind untereinander verkettet, bei vergebenen Speicherbereichen enthält `next` den magischen Wert `0xbaadf00d`

1 malloc-Funktion

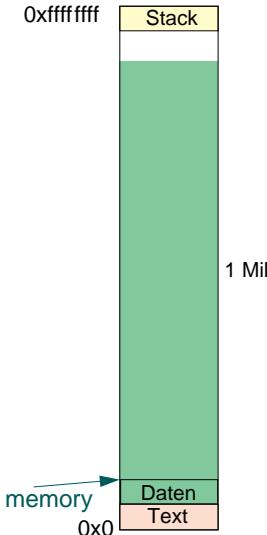
- Beispiel für die Situation nach 3 malloc-Aufrufen (32-Bit-Architektur)



2 malloc-Interna - Initialisierung

- initialer Zustand
 - ◆ Speicher statisch allokiert

```
static char memory[1048576];
```



U4.5

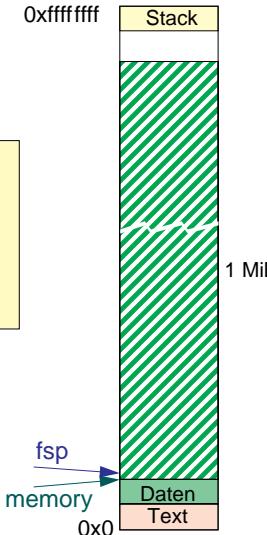
2 malloc-Interna - Initialisierung (2)

- initialer Zustand
 - ◆ Speicher statisch allokiert

```
static char memory[1048576];
```

◆ struct mblock "hineinlegen"

```
// Kopfzeiger der Freispeicherliste
mblock *fsp;
...
fsp = (mblock *) memory;
```



U4.6

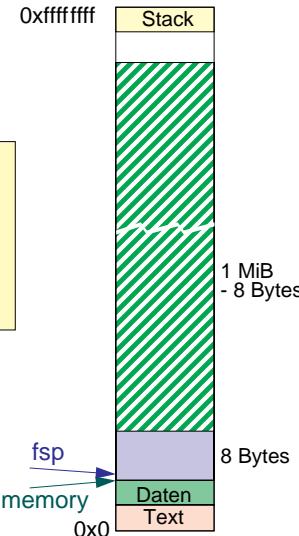
2 malloc-Interna - Initialisierung (3)

- initialer Zustand
 - ◆ Speicher statisch allokiert

```
static char memory[1048576];
```

- ◆ struct mblock "hineinlegen"

```
// Kopfzeiger der Freispeicherliste
mblock *fsp;
...
fsp = (mblock *) memory;
```



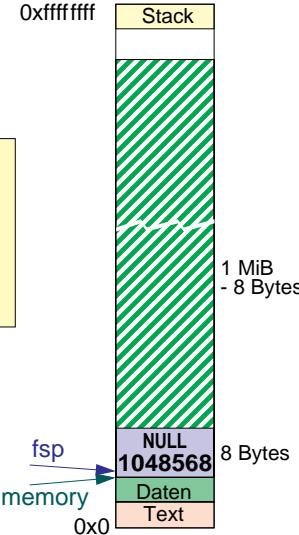
2 malloc-Interna - Initialisierung (4)

- initialer Zustand
 - ◆ Speicher statisch allokiert

```
static char memory[1048576];
```

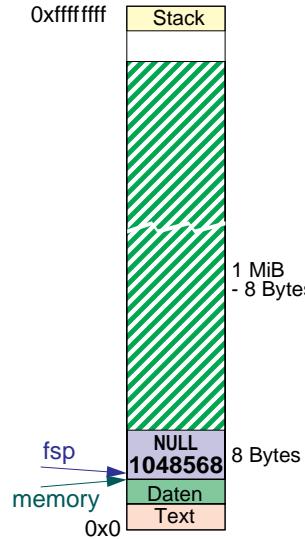
- ◆ struct mblock "hineinlegen"

```
// Kopfzeiger der Freispeicherliste
mblock *fsp;
...
fsp = (mblock *) memory;
fsp->size = sizeof(memory)-sizeof(mblock);
fsp->next = NULL;
```



2 malloc-Interna - Initialisierung (5)

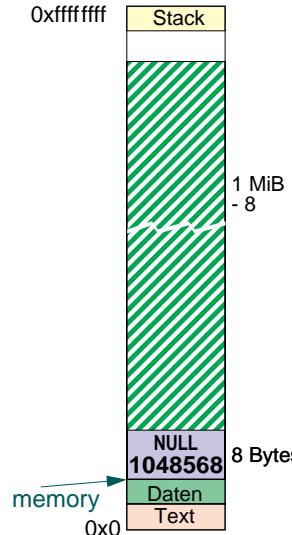
- initialer Zustand
- ➔ zwei Zeiger mit unterschiedlichem Typ zeigen auf den gleichen Speicherbereich
 - unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponentenzugriffe)



3 malloc-Interna - Speicheranforderung

- Aufgaben bei einer Speicheranforderung

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

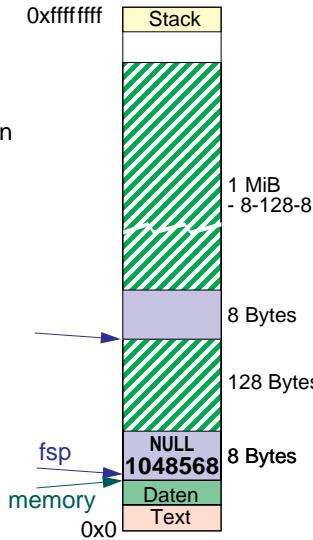


3 malloc-Interna - Speicheranforderung (2)

■ Aufgaben bei einer Speicheranforderung

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

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

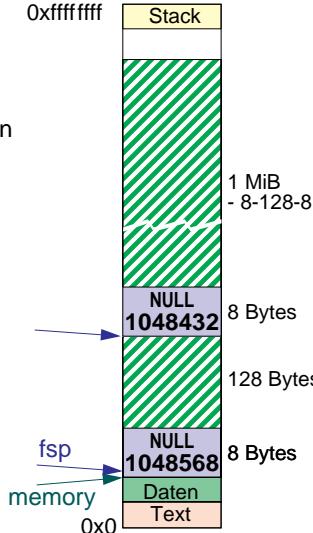


3 malloc-Interna - Speicheranforderung (3)

■ Aufgaben bei einer Speicheranforderung

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

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

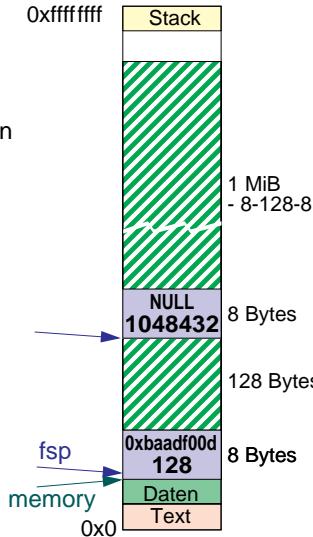


3 malloc-Interna - Speicheranforderung (4)

■ Aufgaben bei einer Speicheranforderung

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

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

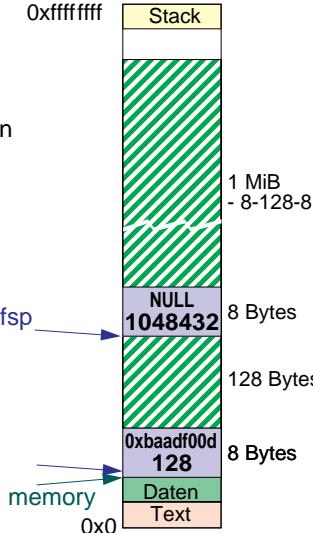


3 malloc-Interna - Speicheranforderung (5)

■ Aufgaben bei einer Speicheranforderung

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

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

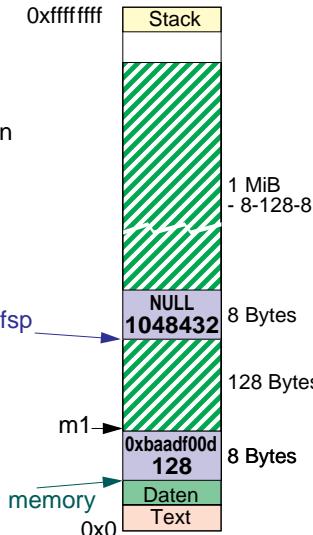


3 malloc-Interna - Speicheranforderung (6)

■ Aufgaben bei einer Speicheranforderung

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

- ◆ 128 Bytes 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 Bytes zurückgeben

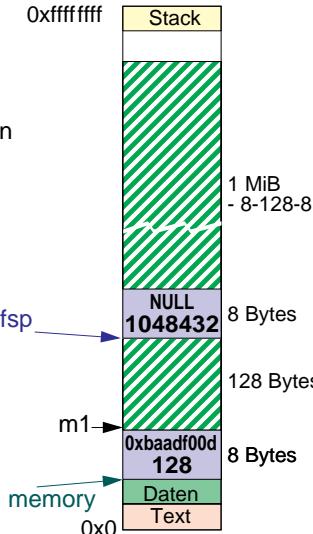


3 malloc-Interna - Speicheranforderung (7)

■ Aufgaben bei einer Speicheranforderung

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

- ◆ 128 Bytes 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 Bytes zurückgeben



4 malloc-Interna - Zeigerarithmetik

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

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

4 malloc-Interna - Zeigerarithmetik (2)

- Variante 2: Berechnungen in mblock-Einheiten

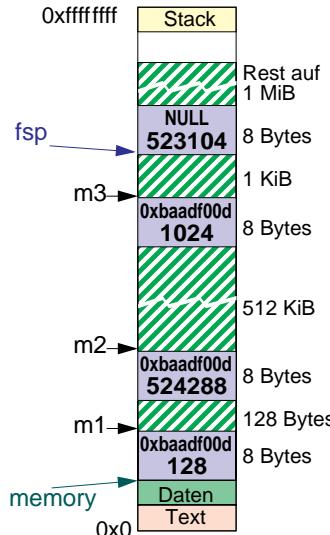
```
void *malloc(size_t size) {
    mblock *fsp_neu, *fsp_alt;
    int units;
    fsp_alt = fsp;
    ...
    units = ( (size-1) / sizeof(mblock) ) + 1;
    fsp_neu = fsp + 1 + units;
```

- ◆ Unterschied: Aufrundung von size auf Vielfaches von sizeof(mblock)
- ◆ Vorteil: die mblock-Strukturen liegen nach einer Anforderung von "krummen" Speichermengen nicht auf "ungeraden" Speichergrenzen
 - manche Prozessoren fordern, dass int-Werte immer auf Wortgrenzen (z.B. 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!

5 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);
...
free(m2);
```

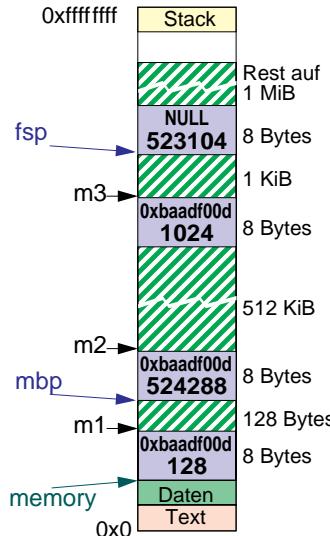


5 malloc-Interna - Speicher freigeben (2)

■ Freigabe von m2 - Aufgaben

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

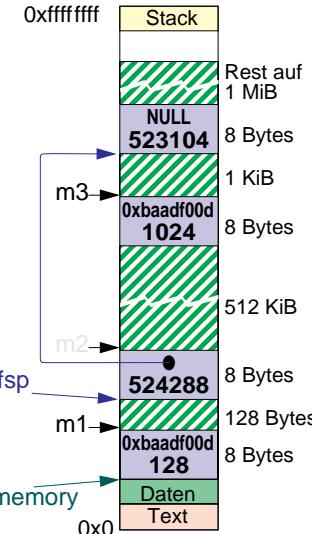
- ◆ Zeiger `mbp` auf zugehörigen `mblock` ermitteln
 - ◆ überprüfen, ob ein gültiger, belegter `mblock` vorliegt (`0xbaa0d00d`)



5 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);
```



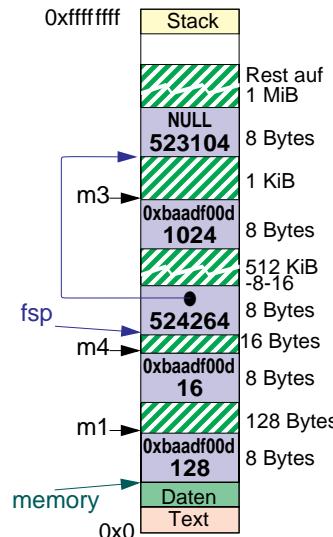
- ◆ Zeiger `mbp` auf zugehörigen mblock ermitteln
- ◆ überprüfen, ob ein gültiger, belegter mblock vorliegt (0xbaadf00d)
- ◆ `fsp` auf freigegebenen Block setzen, bisherigen fsp-mblock verketten

6 malloc-Interna - erneut Speicher anfordern

■ neue Anforderung von 10 Bytes

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

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



7 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 SP2 behandelt
 - (z. B. First-Fit, Best-Fit, Worst-Fit oder Buddy-Verfahren)