U3 3. Übung

- Besprechung der 1. Aufgabe
- Fehlerbehandlung
- Infos zur Aufgabe 3: malloc-Implementierung

U3-1 Fehlerbehandlung

- Fehler können aus unterschiedlichsten Gründen im Programm auftreten
  - Systemressourcen erschöpft
    ⇝ **malloc(3)** schlägt fehl
  - Fehlerhafte Benutzereingaben (z.B. nicht existierende Datei)
    ⇝ **open(2)** schlägt fehl
  - Transiente Fehler (z.B. nicht erreichbarer Server)
    ⇝ **connect(2)** schlägt fehl
  - ...
- Gute Software **erkennt Fehler**, führt eine **angebrachte Behandlung** durch und gibt eine **aussagekräftige Fehlermeldung** aus
  - Kann das Programm trotz des Fehlers sinnvoll weiterlaufen?
  - Beispiel 1: Ermittlung des Hostnamens zu einer IP-Adresse für Logeintrag
    ⇝ Fehlerbehandlung: IP-Adresse im Log eintragen, Programm läuft weiter
  - Beispiel 2: Öffnen einer zu kopierenden Datei schlägt fehl
    ⇝ Fehlerbehandlung: Kopieren nicht möglich, Programm beenden

1 Fehler in Bibliotheksfunktionen

- Fehler treten häufig in Funktionen der C-Bibliothek auf
  - erkennbar i.d.R. am Rückgabewert (Manpage!)
- Die Fehlerursache wird über die globale Variable `errno` übermittelt
  - Fehlercode für jeden möglichen Fehler (siehe **errno(3)**)
  - Der Wert `errno=0` bedeutet Erfolg, alles andere ist ein Fehlercode
  - Bibliotheksfunktionen setzen `errno` im Fehlerfall
  - Bekanntmachung im Programm durch Einbinden von `errno.h`
- Fehlercodes können mit den Funktionen **perror(3)** und **strerror(3)** ausgegeben bzw. in lesbare Strings umgewandelt werden

```
char *mem = malloc(...); /* malloc gibt im Fehlerfall */
if(NULL == mem) {        /* NULL zurück */
    fprintf(stderr, "%s:%d: malloc failed with reason: %s\n",
        __FILE__, __LINE__-3, strerror(errno));
    perror("malloc"); /* Alternative zu strerror + fprintf */
    exit(EXIT_FAILURE); /* Programm mit Fehlercode beenden */
}
```


U3-2 Aufgabe 3: einfache malloc-Implementierung

1 Überblick

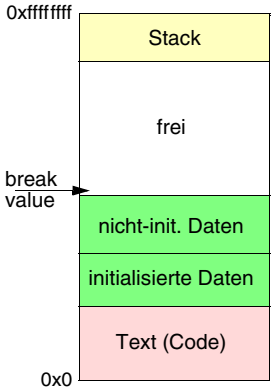
- erheblich vereinfachte Implementierung
  - nur einmal am Anfang Speicher vom Betriebssystem anfordern (1 MB)
  - First-Fit-Allokationsstrategie
  - freigegebener Speicher wird in einer einfachen verketteten Liste verwaltet (benachbarte freie Blöcke werden nicht mehr verschmolzen)
  - `realloc` verlängert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



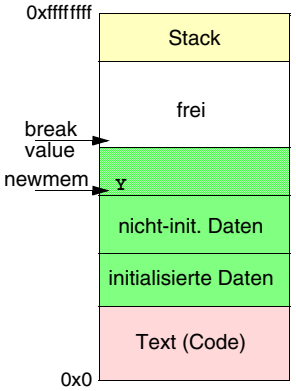
- break value: Ende des Datensegments
- **brk(2)** erlaubt es, diese Adresse neu festzulegen
  - ➤ zusätzlicher Speicher hinter den nicht-initialisierten Daten
- Schnittstellen:

```
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

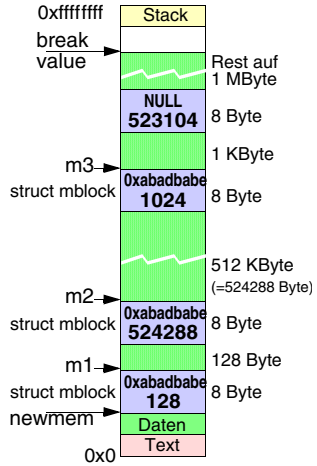
```
struct mblock {
    size_t size;
    struct mblock *next;
};
```

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

4 malloc-Funktion

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

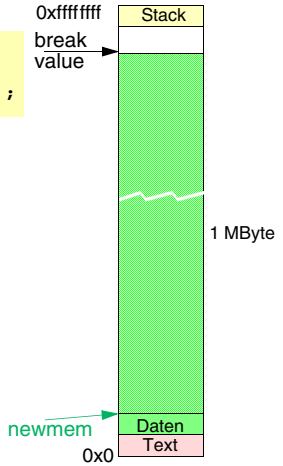
```
...
char *m1, *m2, *m3;
...
m1 = (char *)malloc(128);
m2 = (char *)malloc(512*1024);
m3 = (char *)malloc(1024);
```



5 malloc-Interna - Initialisierung

- initialer Zustand nach sbrk
  - Speicher mit sbrk anfordern

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



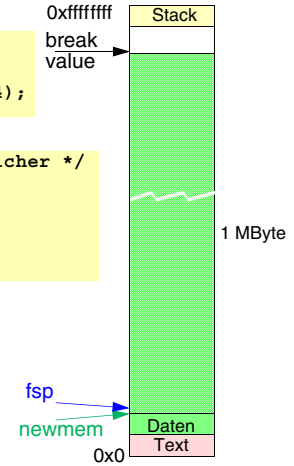
5 malloc-Interna - Initialisierung (2)

- 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;
```



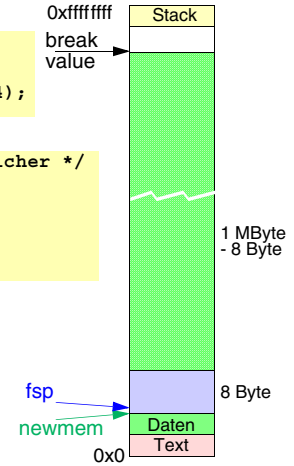
5 malloc-Interna - Initialisierung (3)

- 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;
```



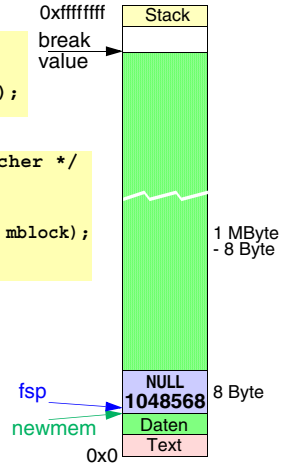
5 malloc-Interna - Initialisierung (4)

- 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

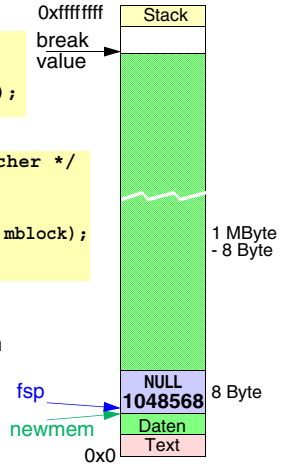
```
char *newmem;
...
newmem = (char *)sbrk(1024*1024);
```

  - 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

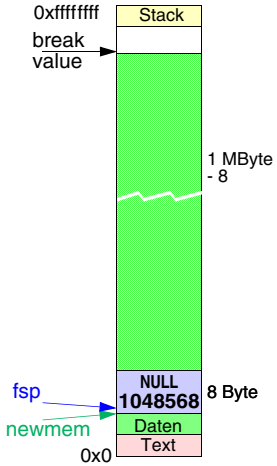
➤ unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponentenzugriffe)



6 malloc-Interna - Speicheranforderung

- Aufgaben bei einer Speicheranforderung

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

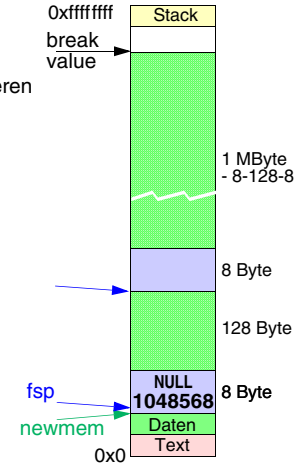


6 malloc-Interna - Speicheranforderung (2)

- Aufgaben bei einer Speicheranforderung

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

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

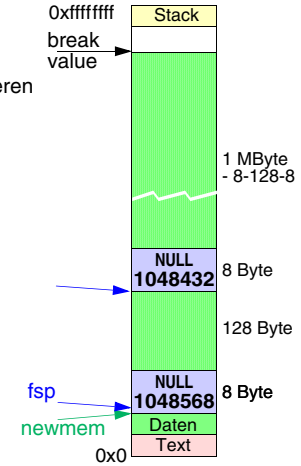


6 malloc-Interna - Speicheranforderung (3)

- Aufgaben bei einer Speicheranforderung

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

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

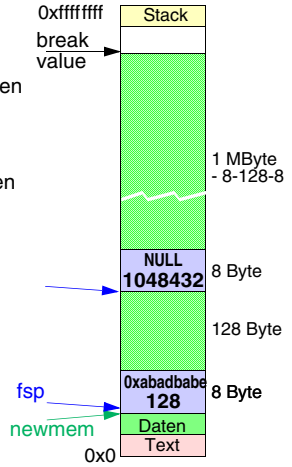


6 malloc-Interna - Speicheranforderung (4)

- 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

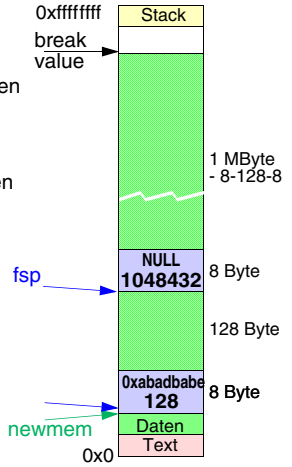


6 malloc-Interna - Speicheranforderung (5)

- 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

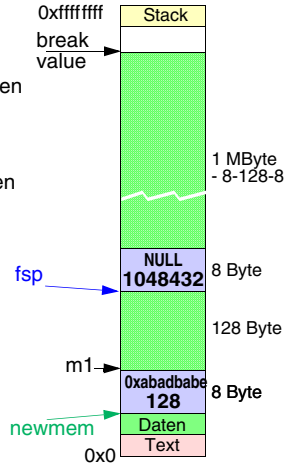


6 malloc-Interna - Speicheranforderung (6)

- 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



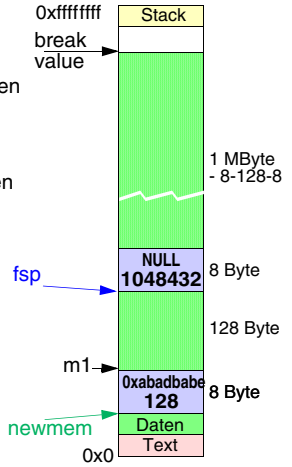
6 malloc-Interna - Speicheranforderung (7)

- 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

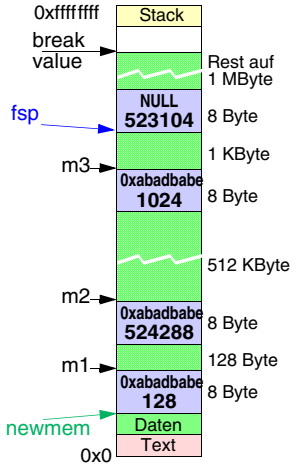
```
void *malloc(size_t size) {
    struct mblock *fsp_neu, *fsp_alt;
    int units;
    fsp_alt = fsp;
    ...
    units = ( (size-1) / sizeof(struct mblock) ) + 1;
    fsp_neu = fsp + 1 + units;
    ...
    return((void *)(fsp_alt + 1));
```

  - ◆ 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 (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!

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);
```

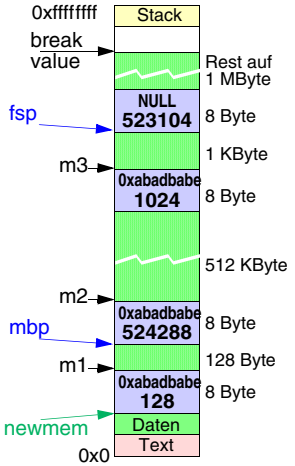


8 malloc-Interna - Speicher freigeben (2)

- 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 (0xabadbabe!)

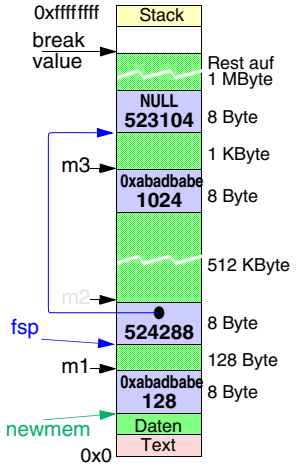


8 malloc-Interna - Speicher freigeben (3)

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

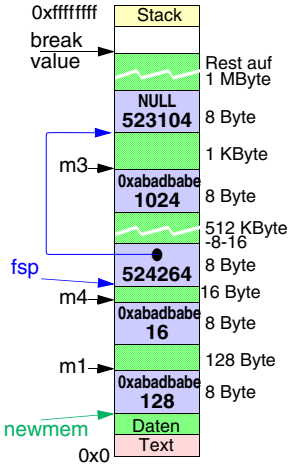


9 malloc-Interna - erneut Speicher anfordern

- 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)