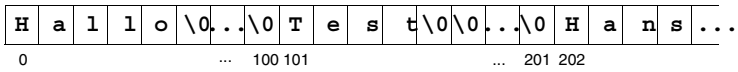
U4 4. Übung

- Besprechung Aufgabe 2 (wsort)
- Aufgabe 4: malloc-Implementierung

U4-1 Aufgabe 2: Sortieren mit qsort

1 wsort - Datenstrukturen (1. Möglichkeit)

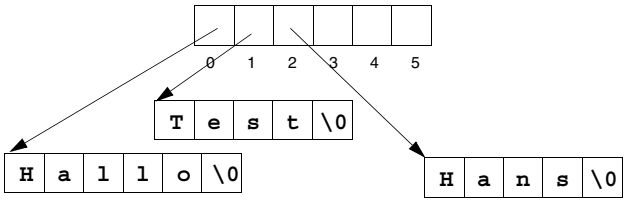
- Array von Zeichenketten



- Vorteile:
  - einfach
- Nachteile:
  - hoher Kopieraufwand (303 Bytes pro Umordnung)
  - maximale Länge der Wörter 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 Bytes 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
  - Vergrößerung aus Effizienzgründen in Blöcken (z.B. je 1000 weitere Elemente)
  - Achtung: realloc kopiert möglicherweise das Array (teuer)
- dynamisch allokierter 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 Zeiger auf die Vergleichsfunktion vom Typ

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

- Lösung: Typcast
  - innerhalb der Funktion, z.B. (Lösungsvariante 2 mit Zeigerarray)
    (weniger schön: Schnittstelle der Funktion wird nach außen verändert)

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

  - schöner, beim qsort-Aufruf:

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


5 wsort-Wettbewerb

- Wer implementiert das schnellste **wsort**?
- Entkoppelt von Aufgabe 2 ☞ eigene Aufgabe **aufgabe0**
  - Projektverzeichnis `/proj/i4sp/$LOGIN/trunk/aufgabe0/`
  - Teilnahme (31.1.2011, 13:37 Uhr): `/proj/i4sp/bin/submit aufgabe0`
  - `Makefile` welches ein Programm `wsort` aus beliebigen weiteren Dateien baut, unter Verwendung eurer gewünschten Compiler-Flags!
- Teilnahme allein oder in Teams beliebiger Größe
- Die schnellste Lösung wird mit einem Kasten Zirndorfer prämiert!
- Ideenaustausch und Bekanntgabe von Zwischenzeiten im Forum
  ☞ `http://fsi.informatik.uni-erlangen.de/forum`

5 wsort-Wettbewerb: Regeln

- Vollständige Funktionalität entsprechend Aufgabenstellung 2
- Korrektheit bei allen Eingaben
- Compilerflags sind frei wählbar
- Zeitmessung
  - Referenzrechner: faui06[a-o] (Core 2 Quad Q6600, 8 GB RAM)
  - Pro Wortliste werden mehrere Durchläufe in Reihe gestartet, die kürzeste Zeit gilt
  - Relevant ist die **Summe der Real-Zeiten** für **wlist5** und **wlist6**
  - Testen mit: `/proj/i4sp/pub/aufgabe0/runtest.sh`

5 wsort-Wettbewerb: Ansatzpunkte für Speedups

- Compiler-Flags
  Optimierungsstufe, Zielarchitektur
- Einlesen und Ausgeben
  Anzahl der Lese- bzw. Schreiboperationen
- Speicherverwaltung
  Wenige Kopieraktionen bzw. Reallokierungen
- Parallelisierung
  Arbeit auf mehrere Threads verteilen
- Sortierverfahren
  qsort(3) ist nicht langsam, aber nicht für alle Zwecke optimal

U4-2 Aufgabe 4: einfache malloc-Implementierung

- First-Fit-Allokationsstrategie
- Vereinfachungen:
  - 1 MiB Speicher statisch allokiert
  - freier Speicher wird in einer einfach-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
- Ziele der Aufgabe
  - Zusammenhang zwischen "nacktem Speicher" und typisierten Datenbereichen verstehen
  - Funktion aus der C-Bibliothek selbst realisieren

1 malloc-Funktion

- 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

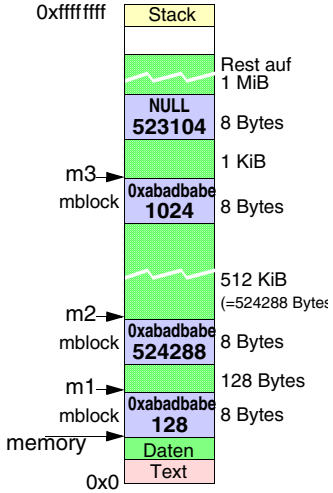
```
typedef struct mblock {
    size_t size;          // Größe des anhängenden Bereichs
    struct mblock *next; // Verkettung freier Bereiche
} mblock;
```

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

1 malloc-Funktion

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

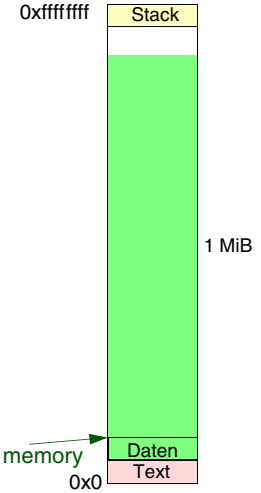
```
...
char *m1, *m2, *m3;
...
m1 = (char *) malloc(128);
m2 = (char *) malloc(512*1024);
m3 = (char *) malloc(1024);
```



2 malloc-Interna - Initialisierung

- initialer Zustand
  - Speicher statisch allokiert

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



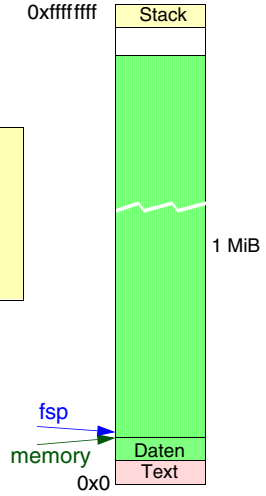
2 malloc-Interna - Initialisierung (2)

- initialer Zustand
  - Speicher statisch allokiert

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

  - struct mblock "hineinlegen"

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



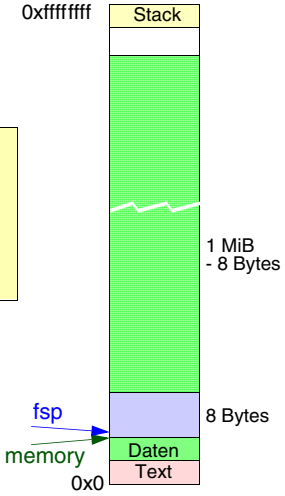
2 malloc-Interna - Initialisierung (3)

- initialer Zustand
  - Speicher statisch allokiert

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

  - struct mblock "hineinlegen"

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



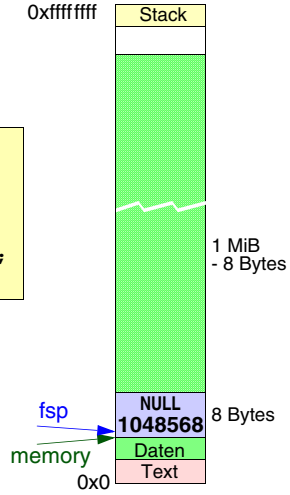
2 malloc-Interna - Initialisierung (4)

- 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;
```

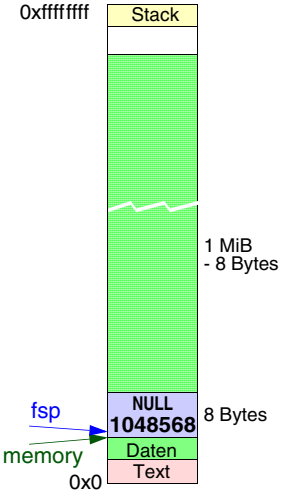


2 malloc-Interna - Initialisierung (5)

- initialer Zustand

➥ zwei Zeiger mit unterschiedlichem Typ zeigen auf den gleichen Speicherbereich

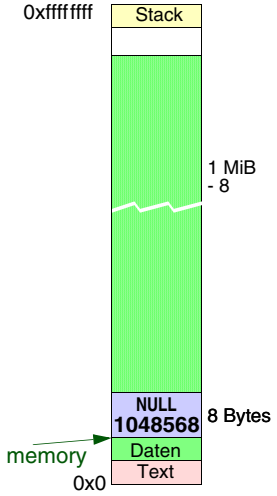
➤ unterschiedliche Semantik beim Zugriff (Zeigerarithmetik, Strukturkomponentenzugriffe)



3 malloc-Interna - Speicheranforderung

- Aufgaben bei einer Speicheranforderung

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

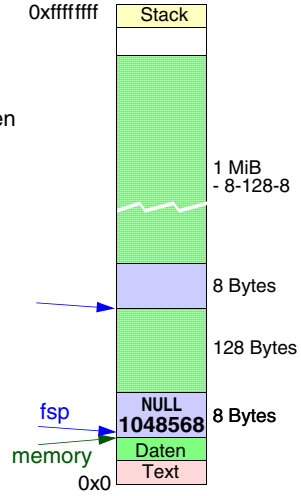


3 malloc-Interna - Speicheranforderung (2)

- Aufgaben bei einer Speicheranforderung

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

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

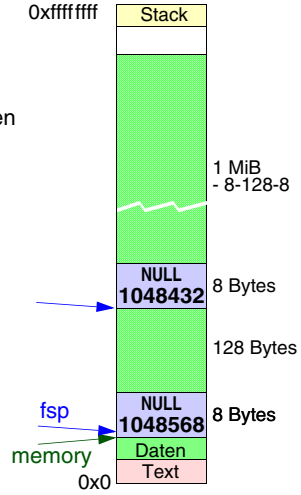


3 malloc-Interna - Speicheranforderung (3)

- Aufgaben bei einer Speicheranforderung

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

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

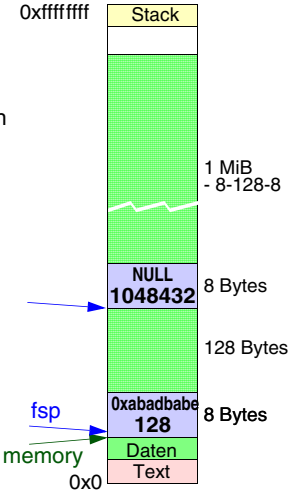


3 malloc-Interna - Speicheranforderung (4)

- 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

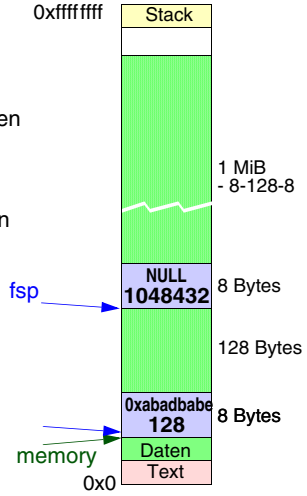


3 malloc-Interna - Speicheranforderung (5)

- 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

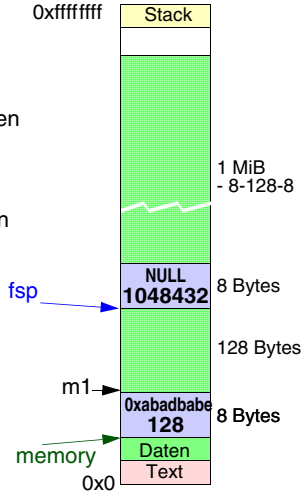


3 malloc-Interna - Speicheranforderung (6)

- 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



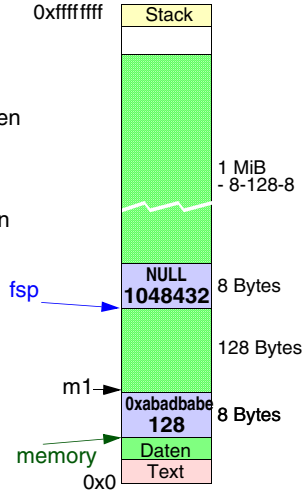
3 malloc-Interna - Speicheranforderung (7)

- 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

- Frage:
  wie rechnet man auf dem Speicher?
  - ➤ in char?
  - ➤ in mblock?



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);
    ...
    return((void *)(fsp_alt + 1));
```

4 malloc-Interna - Zeigerarithmetik (2)

- Variante 2: Berechnungen in mblock-Einheiten

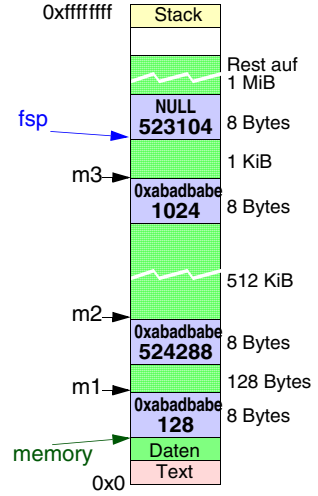
```
void *malloc(size_t size) {
    mblock *fsp_neu, *fsp_alt;
    int units;
    fsp_alt = fsp;
    ...
    units = ( (size-1) / sizeof(mblock) ) + 1;
    fsp_neu = fsp + 1 + units;
    ...
    return((void *)(fsp_alt + 1));
```

  - 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);
```

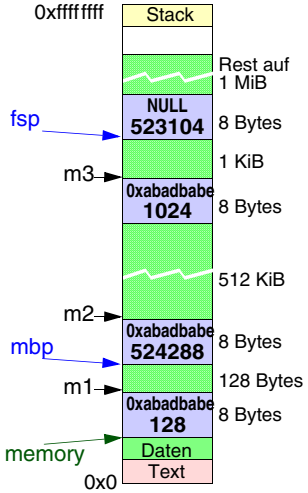


5 malloc-Interna - Speicher freigeben (2)

- 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 (0xabadbabe)

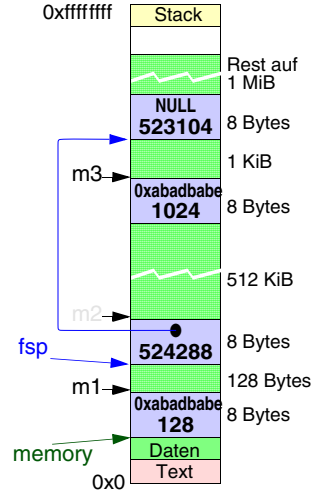


5 malloc-Interna - Speicher freigeben (3)

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

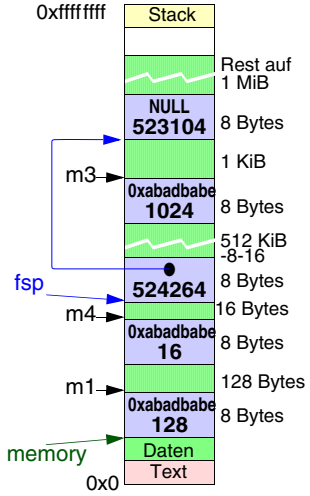


6 malloc-Interna - erneut Speicher anfordern

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