# Übung 4, Aufgabe 4: Implementierung einer rudimentären Speicherverwaltung in C

Paul Hänsch

23. November 2009

# Inhalt

# Framework



```
#define memorySize 10240

enum memBlockState{not_allocated=0, allocated=1};

typedef struct _memoryblock {
        void* data;
        int dataLength;
        enum memBlockState state;
        struct _memoryblock * nextBlock;
} memoryBlock;

#define memoryBlockHeaderSize sizeof(memoryBlock)

char memory[memorySize];
```

# Mein Ansatz



Jedem Speicherblock wird ein einzelner signierter Integer vorangestellt, der den freien Speicher in dem jeweiligen Block anzeigt.

- negativer Wert für belegten Block
- Blockadresse wird aus Größe des Vorgängerblocks errechnet
- komplexere Datenstrukuren nicht benötigt

Nachteile:

- 1 Bit weniger für Speicheradressen
- keine Aufnahme weiterer Daten möglich

# Mein Ansatz



Jedem Speicherblock wird ein einzelner signierter Integer vorangestellt, der den freien Speicher in dem jeweiligen Block anzeigt.

- negativer Wert für belegten Block
- Blockadresse wird aus Größe des Vorgängerblocks errechnet
- komplexere Datenstrukuren nicht benötigt

Nachteile:

- 1 Bit weniger für Speicheradressen
- keine Aufnahme weiterer Daten möglich

# Mein Ansatz



Jedem Speicherblock wird ein einzelner signierter Integer vorangestellt, der den freien Speicher in dem jeweiligen Block anzeigt.

- negativer Wert für belegten Block
- Blockadresse wird aus Größe des Vorgängerblocks errechnet
- komplexere Datenstrukuren nicht benötigt

Nachteile:

- 1 Bit weniger für Speicheradressen
- keine Aufnahme weiterer Daten möglich

# Mein Ansatz



Jedem Speicherblock wird ein einzelner signierter Integer vorangestellt, der den freien Speicher in dem jeweiligen Block anzeigt.

- negativer Wert für belegten Block
- Blockadresse wird aus Größe des Vorgängerblocks errechnet
- komplexere Datenstrukuren nicht benötigt

Nachteile:

- 1 Bit weniger für Speicheradressen
- keine Aufnahme weiterer Daten möglich

# Mein Ansatz



Jedem Speicherblock wird ein einzelner signierter Integer vorangestellt, der den freien Speicher in dem jeweiligen Block anzeigt.

- negativer Wert für belegten Block
- Blockadresse wird aus Größe des Vorgängerblocks errechnet
- komplexere Datenstrukuren nicht benötigt

Nachteile:

- 1 Bit weniger für Speicheradressen
- keine Aufnahme weiterer Daten möglich

# Mein Ansatz



Jedem Speicherblock wird ein einzelner signierter Integer vorangestellt, der den freien Speicher in dem jeweiligen Block anzeigt.

- negativer Wert für belegten Block
- Blockadresse wird aus Größe des Vorgängerblocks errechnet
- komplexere Datenstrukuren nicht benötigt

Nachteile:

- 1 Bit weniger für Speicheradressen
- keine Aufnahme weiterer Daten möglich

# Mein Ansatz



Jedem Speicherblock wird ein einzelner signierter Integer vorangestellt, der den freien Speicher in dem jeweiligen Block anzeigt.

- negativer Wert für belegten Block
- Blockadresse wird aus Größe des Vorgängerblocks errechnet
- komplexere Datenstrukuren nicht benötigt

Nachteile:

- 1 Bit weniger für Speicheradressen
- keine Aufnahme weiterer Daten möglich

# Zum Verständnis...

# Breite von Datentypen



Wie breit ist ein Integer?

# Breite von Datentypen



Wie breit ist ein Integer?

- 4 Byte 

?

# Breite von Datentypen



Wie breit ist ein Integer?

- 4 Byte 

?
- 8 Byte 

??

# Breite von Datentypen



Wie breit ist ein Integer?

- 4 Byte 

?
- 8 Byte 

??
- 16 Byte 

???

# Breite von Datentypen



Wie breit ist ein Integer?

- 4 Byte 

?
- 8 Byte 

??
- 16 Byte 

???

Ein Integer ist immer sizeof(int) breit.

# Rechnen mit Pointern



Code:

```
int main(){
  int *foo = NULL;
  printf("Foo-Pointer: %i\n", foo);
  foo = foo + 1;
  printf("Foo-Pointer: %i\n", foo);
  return 0;
}
```

# Rechnen mit Pointern



Ausführung:

```
picus% gcc test.c -o test; ./test
Foo-Pointer: 0
Foo-Pointer: 4
```

# Rechnen mit Pointern



**BLACK MAGICK?**

# Rechnen mit Pointern



**BLACK MAGICK?**

- Black mages cast spells, C casts data types!

```
foo = (int*) foo + (int*) 1;
```

# Rechnen mit Pointern



**BLACK MAGICK?**

- Black mages cast spells, C casts data types!

```
foo = (int*) foo + (int*) 1;
```

- Was tut man dagegen?

```
foo = (int*)((int) foo + 1);
```

# Programm



```
#include <stdio.h>

int memsize = 10240;
int memory[10240 / sizeof(int)];

int *nextBlock(int *blk){
  if (blk == NULL) return memory;
  int size = *blk;

  if (size < 0) size *= -1;
  blk = (int*)((int) blk + size) + 1;

  if (blk < (int*)((int)memory + memsize) - 1) return blk;
  else return NULL;
}

int *bestFit(int size){
  int *best = NULL, *blk = NULL;

  while (blk = nextBlock(blk)){
    if ((*blk - size) >= 0 && (best == NULL || (*blk - size) < (*best - size))) best = blk;
  }

  return best;
}

void *my_malloc(int size){
  if (!*memory) *memory = memsize - sizeof(int);
  int oldblk, *blk = bestFit(size);
  if (blk == NULL) return NULL;
  else{
```

# Programm



**Dont Panic!**

# Blockiteration



```
int *nextBlock(int *blk){
  if (blk == NULL) return memory;
  int size = *blk;

  if (size < 0) size *= -1;
  blk = (int*)((int) blk + size) + 1;

  if (blk < (int*)((int)memory + memsize) - 1)
    return blk;
  else return NULL;
}
```

# Best Fit



```
int *bestFit(int size){
  int *best = NULL, *blk = NULL;

  while (blk = nextBlock(blk)){
    if ((*blk - size) >= 0 &&
        (best == NULL ||
         (*blk - size) < (*best - size))
       ) best = blk;
  }

  return best;
}
```

# Malloc



```
void *my_malloc(int size){
  if (!*memory) *memory = memsize - sizeof(int);
  int oldblk, *blk = bestFit(size);
  if (blk == NULL) return NULL;
  else{
    oldblk = *blk;
    *blk = (size * -1);
    *nextBlock(blk) = oldblk - size - sizeof(int);
  }
  return blk + 1;
}
```

# Free



```
void my_free(int *blk){
  blk -= 1;
  *blk = (*blk * -1);
  defrag();
}
```

# Defrag



```
void defrag(){
  int *blk = NULL;
  while (blk = nextBlock(blk)){
    if (*blk > 0)
      while (nextBlock(blk) && *nextBlock(blk) > 0)
        *blk += *nextBlock(blk) + sizeof(int);
  }
}
```

...