

Programmiertechnik II

Freispeicherverwaltung

Speicherblöcke

- Objektzustand ist in Speicherblöcken dargestellt
 - ein Block oder mehrere Blöcke
- Jeder Block hat eine Anfangsadresse und eine Länge
 - Länge ist u.U. nur implizit bekannt, nicht explizit gespeichert
- Ziel der Speicherverwaltung
 - Bereitstellung von Speicher für Objektzustand
 - Wiederverwendung von Speicher von Objekten, die freigegeben wurden
 - Freispeicherverwaltung: Wiederverwendung von freigegebenem Speicher
 - Garbage Collection: Freigabe von nicht mehr verwendeten Objekten

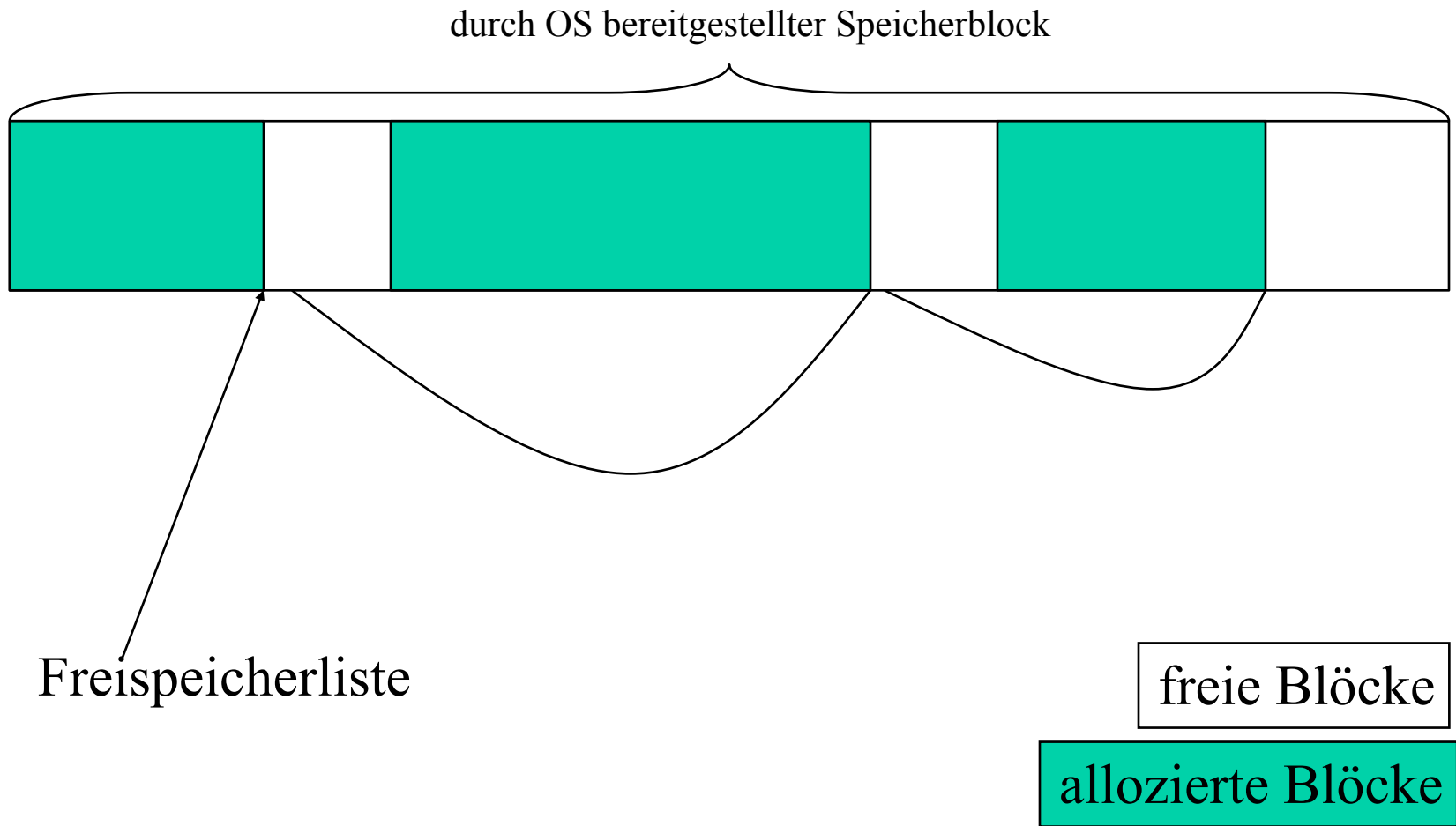
Schnittstellen

- **Allozierung: Bereitstellung von freiem Speicher**
 - Rückgabe: Pointer auf Anfang von Speicher
 - Unter Angabe der Größe: `void *malloc(size_t requested);`
 - Mit impliziter Angabe der Größe: `new Foo();`
 - Objekte einer Klasse haben alle die gleiche Größe
 - `new`-Ausdruck in C++ alloziert Speicher, führt dann Konstruktor aus
 - Alignment: Ergebnisadresse muss “gerade” sein (etwa: durch 8 teilbar)
- **Deallozierung: Freigabe**
 - Eingabe: Pointer auf Anfang des Speicherblocks
 - u.U. zusätzlich Größe des Speicherblocks
 - Operation ohne Ergebniswert, u.U. Überprüfung, ob Speicher tatsächlich alloziert ist
 - `void free(void *data);`

Freispeicherlisten

- Freispeicherverwaltung alloziert großen Speicherblock vom Betriebssystem
 - sbrk(), VirtualAlloc → Betriebssystemvorlesung
- Speicher wird in kleinere Stücken zerlegt und der Anwendung bereitgestellt
 - Information nötig, wieviel Speicher noch frei ist
- Fragmentierung: Freigabe von Speicherblöcken “außer der Reihe”

Fragmentierung



Allokator für Objekte Fester Größe

- Annahme: alle Objekte haben eine feste Größe von 40 Byte
- Annahme: Gesamtspeicher liegt in einem Block (Arena) vor
 - Erweiterung auf mehrere Blöcke wird zunächst nicht betrachtet
- Arena wird in aufeinanderfolgende Blöcke von je 40 Byte unterteilt
- Blöcke werden durchnummeriert
- Für jeden Block wird ein Bit benötigt, um festzuhalten, ob der Block alloziert ist
 - Bits werden hintereinander in Bitmap gespeichert, etwa 16 Bit pro “unsigned short”
 - Speicherbedarf für die Bitmap: Zahl der Blöcke/8 Bytes
- Ausgangszustand: Alle Blöcke sind frei, in der Bitmap haben alle Bits den Wert 0

Allozierung

- Auffinden eines freien Bits in der Bitmap:
 - Schleife über alle Worte in der Bitmap
 - Schleife über alle Bits in einem Wort
 - Wenn das Bit nicht gesetzt ist, ist der Block frei:
 - » Bit setzen, Adresse des Blocks zurückgeben
 - Wenn kein freies Bit gefunden wurde, ist der Speicher erschöpft
- Bit-Operationen in C:
 - $1 \ll n$: Erzeugen einer Bitmaske mit genau einem gesetzten Bit an Position n ($= 2^n$)
 - $x | y$: Bitweises Oder
 - $x \& y$: Bitweises Und
 - $x \wedge y$: Bitweises Exklusiv-Oder
 - $\sim x$: Bitweise Negation (Einerkomplement)

Deallozierung

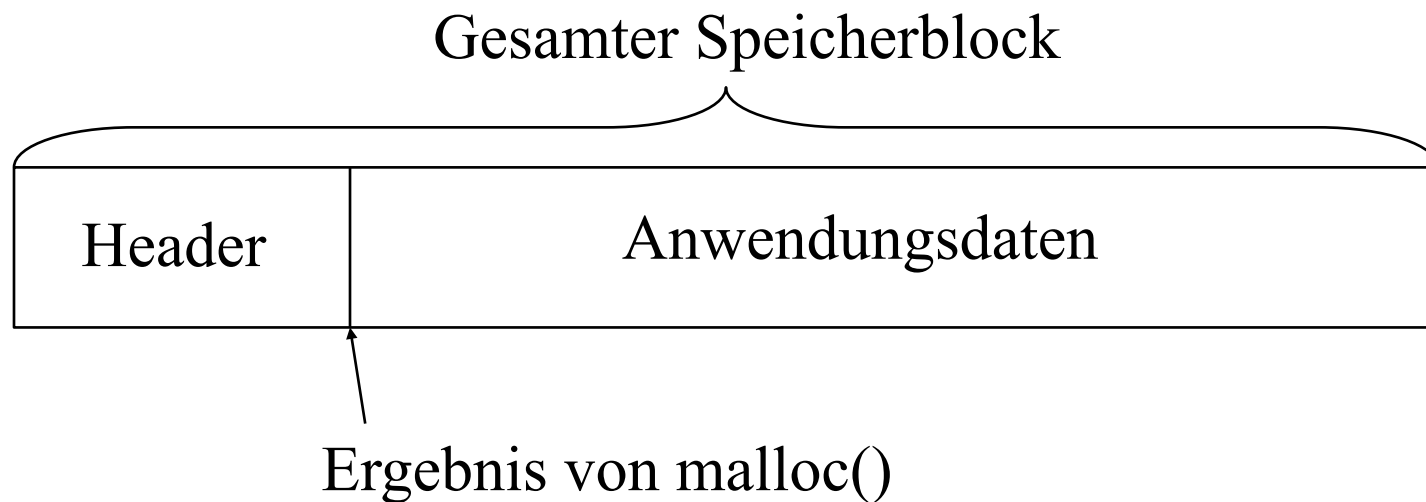
- Ermitteln der Blocknummern
- Ermitteln des Worts, in dem das Bit für den Block gespeichert ist
- Ermitteln des Bits innerhalb des Worts
- Löschen des Bits (Freigabe des Blocks)
- Optional:
 - Test, ob freigegebener Block einer gültigen Blocknummer entspricht
 - Test, ob Blocknummer tatsächlich alloziert ist

Darstellung von Freispeicherlisten

- Schnittstelle:
 - `void* malloc(size_t requested);`
 - `void free(void *data);`
- Problem: Größe des Speicherblocks wird bei Freigabe nicht angegeben
 - Speicherverwaltung muss Größe des Blocks separat speichern
 - Idee: Blockgröße wird unmittelbar vor dem Speicherblock abgelegt
- Problem: Freispeicherverwaltung benötigt Speicher zur Darstellung der Listenverkettung
 - Idee: Freispeicherblock enthält Listenverkettung selbst
- Liste von allozierten Speicherblöcken?
 - Teuer, da Listenverkettung weiteren Speicher benötigt
 - unnötig, da Anwendung allozierten Speicher explizit freigeben muss

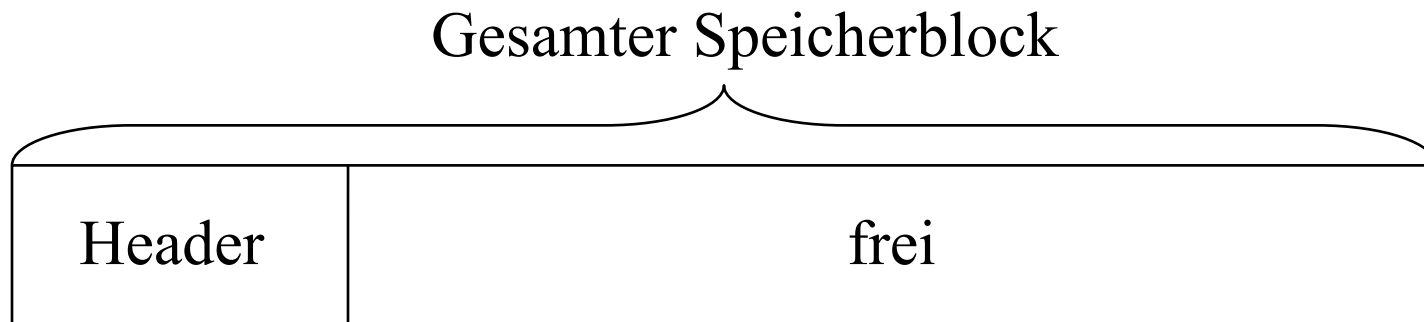
Struktur eines allozierten Speicherblocks

- Header vor eigentlichen Daten
- Anwendung “sieht” nur Anfang der eigenen Daten



Struktur eines Freispeicherblocks

- Jeder Block enthält Größe und Verweis auf nächsten Block
 - auf 32-bit System 4 Byte Größe, 4 Byte Verweis auf nächsten Block
 - minimale Blockgröße 8 Byte



Header-Definition

```
struct AllocatedHeader{  
    int size;  
};
```

```
struct FreeHeader{  
    int size;  
    struct FreeHeader* next;  
};
```

- Alignment: Anfangsadresse des Datenblocks muss evtl. durch 8 teilbar sein
 - es sind auch für den allozierten Block 8 Byte Header üblich

Pointer-Arithmetik

- **Felder (arrays):**
int items[100];
items[10] = 5; // 11. Element, Byte-Offset 10*sizeof(int)
char *pitems = items;
pitems[10] = 5; // desgleichen
- **Pointer-Addition: $p[i]$ ist $*(p+i)$**
 - Pointer-Arithmetik rechnet in Einheiten der Elementgröße
pitems = items + 10; // pitems = &items[10];
pitems[-1] = 7; // items[9] = 7
- **Pointer-Subtraktion: Zahl der Elemente**
int index = pitems - items; // 10

Allokation

- Gegeben: Größe des angeforderten Speichers
- Gesucht: Speicherblock, der mindestens so groß ist wie der geforderte Block
 - Allokierungsstrategie?
 - Zerlegung eines großen Blocks in allozierten Block und Restblock
- Rückgabe des Blocks: Adresse nach dem Header

```
void* malloc(size_t requested){  
    Header *found = ...;  
    found->size = requested;  
    return found+1;  
}
```

Deallokation

- Eintragen des Blocks in Freispeicherliste
 - Zurückrechnen auf Anfang des Headers

```
void free(void *data){  
    Header* hdata = (Header*)data - 1;  
    // naives Eintragen in Liste  
    hdata->next = freelist;  
    freelist = hdata;  
}
```
- Fragmentierung: kleinere hintereinanderliegende Blöcke müssen wieder zu größeren Blöcken verschmolzen werden
 - Freispeicherliste muss aufsteigend nach Adressen sortiert sein
 - Verschmelzung muss sowohl mit dem folgenden als auch mit dem vorhergehenden Block möglich sein
 - doppelt verkettete Liste?

Weitere Aspekte

- Effizientes Auffinden eines freien Speicherblocks passender Größe
 - Allokierungsstrategien
 - Pool-Allokatoren
- Gleichzeitige Manipulation der Freispeicherliste von mehreren Threads
 - Synchronisation; Schutz der Freispeicherliste
 - Verwaltung von separaten Listen für verschiedene Threads