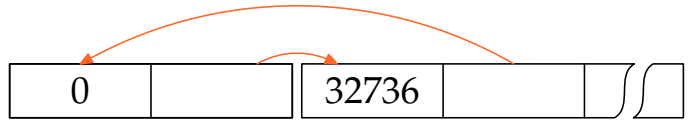
- Im folgenden wird eine sehr einfache Speicherverwaltung vorgestellt, die
  - das Belegen und Freigeben von Speicher unterstützt,
  - freigegebene Speicherflächen wieder zur Verfügung stellen kann und auch
  - in der Lage ist, mehrere hintereinander freigegebene Speicherflächen zu einer größeren Freifläche zusammenzufügen, um Fragmentierung zu vermeiden.
- Der vorgestellte Algorithmus ist in der Literatur bekannt als *circular first fit*. Eine ähnliche Fassung wurde bereits im ersten C-Buch von Kernighan und Ritchie vorgestellt.

alloc.c

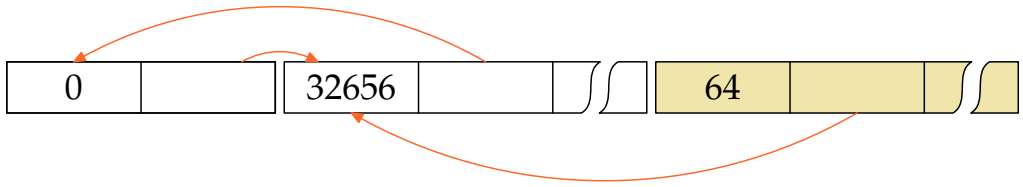
```
typedef struct memnode {
   size_t size;
   struct memnode* next;
} memnode;
```

- Allen freien oder belegten Speicherflächen geht ein Verwaltungsobjekt des Typs *memnode* voraus.
- *size* gibt die Größe der Speicherfläche an, die diesem Verwaltungsobjekt unmittelbar folgt, jedoch ohne Einberechnung des Speicherbedarfs für das Verwaltungsobjekt
- *next* verweist
  - bei freien Speicherflächen auf das nächste Verwaltungsobjekt im Ring freier Speicherflächen
  - bei belegten Speicherflächen zum unmittelbar vorangehenden Speicherelement, egal ob dieses frei oder belegt ist.

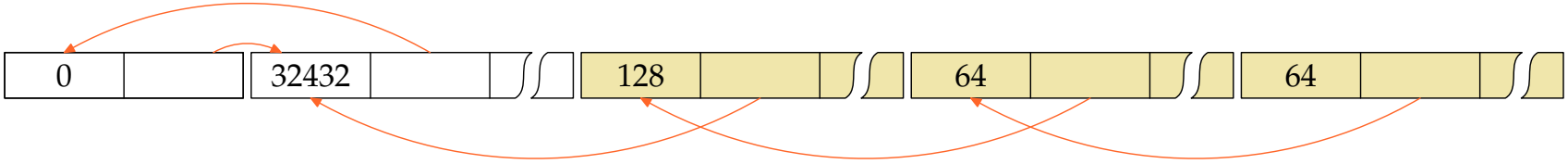


- Alle freien Speicherflächen sind in einem Ring organisiert.
- Ein Ring ist bei dem gewählten Algorithmus *circular first fit* notwendig, weil nicht immer von Anfang an gesucht wird, sondern dort die Suche begonnen wird, wo sie zuletzt endete.
- Damit sich der Ring nicht auflöst, wenn alle zur Verfügung stehenden Speicherflächen vergeben sind, gibt es ein spezielles Ring-Element, das nur aus einem Verwaltungsobjekt besteht, aber keinen eigentlichen Speicherplatz anbietet.
- Das Diagramm zeigt zwei Ringelemente. Links ist das spezielle Element (mit $size = 0$) und rechts ein Element, das noch 32736 Bytes frei hat.

Belegung der ersten Speicherfläche

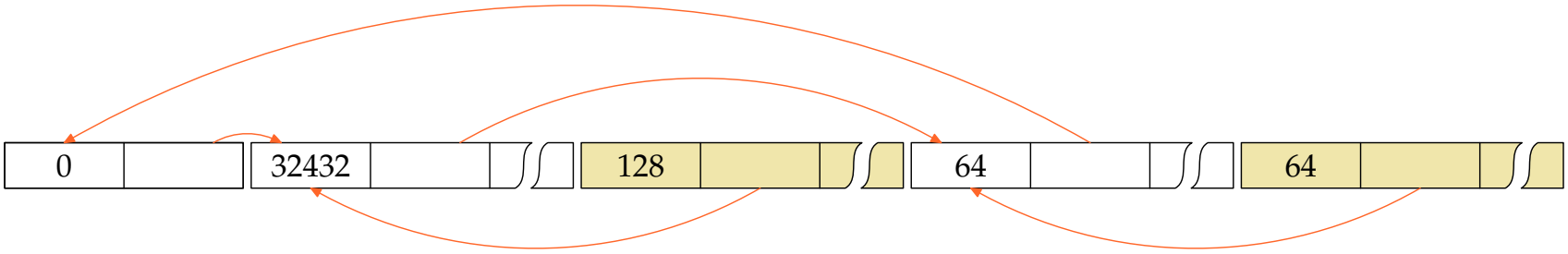


- Wenn nun 64 Bytes belegt werden sollen, wird nach einem Element im Ring der freien Speicherflächen gesucht, das mindestens 64 Bytes anbietet. In diesem Beispiel gab es nur das ganz große Element.
- Da noch Speicher übrigbleibt, wird das Element geteilt: Das Ende wird für die zu vergebende Speicherfläche Platz reserviert mitsamt einem Verwaltungsobjekt und bei dem entsprechenden freien Element wird *size* verkleinert, von 32736 auf 32656 (unter der Annahme, dass ein Verwaltungsobjekt 16 Bytes belegt und somit $64 + 16 = 80$ Bytes benötigt wurden).
- Bei dem Verwaltungsobjekt für die belegte Speicherfläche verweist der *next*-Zeiger auf das im Speicher unmittelbar davorliegende Element; das ist hier noch das Element aus dem Ring der freien Speicherflächen.



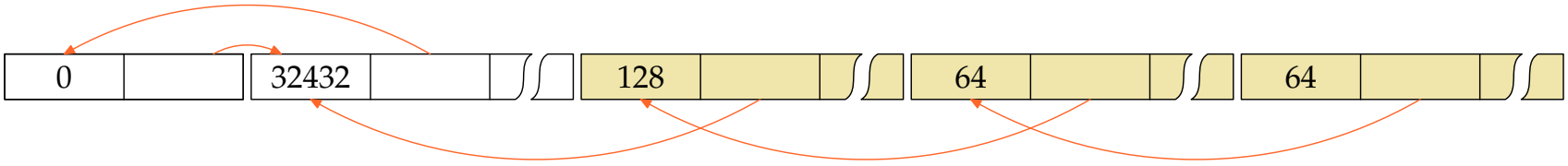
- Inzwischen wurden zwei weitere Speicherflächen belegt, zuerst noch einmal mit 64, dann mit 128 Bytes.
- Diese wurden allesamt dem großen Ringelement der freien Speicherflächen entnommen.
- Zu beachten ist, dass die Verwaltungsobjekte der belegten Speicherflächen jeweils auf das im Speicher unmittelbar davorliegende Element verweisen, egal ob dies frei ist oder nicht. Auf diese Weise ist es bei einer Freigabe recht einfach, ein freigegebenes Element mit einem bereits freien Element in der unmittelbaren Nachbarschaft zu vereinigen.

Freigabe einer Speicherfläche



- Wenn eine belegte Speicherfläche freigegeben wird, wird ausgehend von dem freiwerdenden Element solange die Kette der davorliegenden Elemente verfolgt, bis das erste Ring-Element vorgefunden wird, das eine freie Speicherfläche repräsentiert.
- Die freigegebene Speicherfläche wird unmittelbar dahinter eingehängt.
- Bei dem Ring der freien Speicherflächen bleibt so immer die Ordnung entsprechend der Lage im Speicher erhalten. Nur auf diese Weise ist es später möglich, benachbarte freie Flächen wieder zu größeren freien Flächen zusammenzulegen.

Wie wird eine freie Speicherflächer erkannt? 239



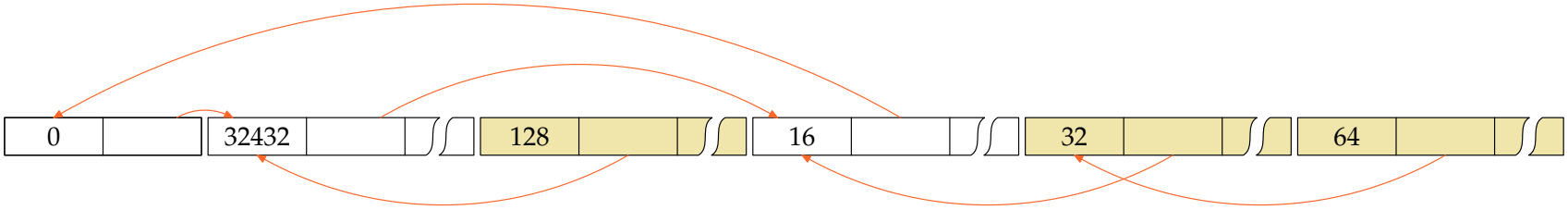
- Wenn bei einer Freigabe das nächste vorangehende freie Element gesucht wird, müssen wir unterscheiden können, ob ein Element frei oder belegt ist.
- Eine Möglichkeit wäre es, etwa bei *size* das niedrigstwertige Bit entsprechend zu setzen. Die Größe muss immer die Alignment-Anforderungen berücksichtigen und entsprechend darf eine Größe nie ungerade sein.
- In diesem einfachen Beispiel mit nur einem großen Speicherblock und einem Spezialelement geht es aber auch ohne diesen Trick...

Wie wird eine freie Speicherflächer erkannt? 240

alloc.c

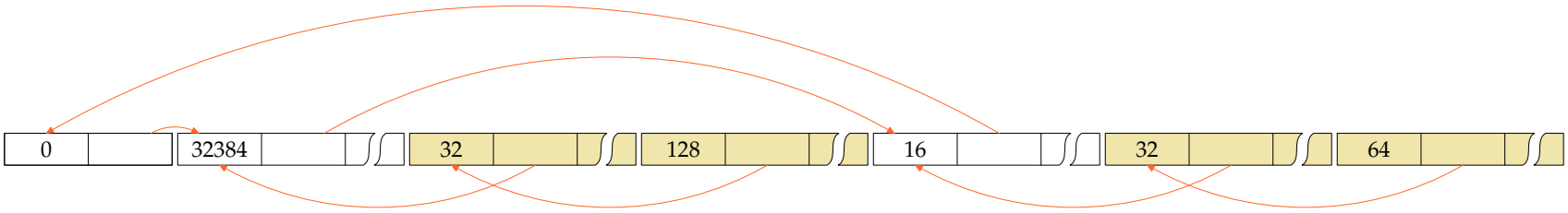
```
bool is_allocated(memnode* ptr) {
   if (ptr == root) return false;
   if (ptr->next > ptr) return false;
   if (ptr->next != root) return true;
   if (root->next > ptr) return true;
   return root->next == root;
}
```

- Das Spezialelement (nennt sich hier *root*) ist immer frei.
- Wenn der Zeiger nach vorne (d.h. zu einer größeren Adresse) weist, dann ist das Element frei.
- Wenn das alles nicht zutrifft und der Zeiger nicht auf das Spezialelement *root* verweist, ist es belegt.
- Wenn *next* auf das Spezialelement *root* verweist, gibt es zwei Fälle:
  - Es ist belegt und das Element liegt unmittelbar hinter dem Spezialelement (am Anfang des großen Blocks) oder
  - es handelt sich um das freie Element mit der höchsten Adresse.

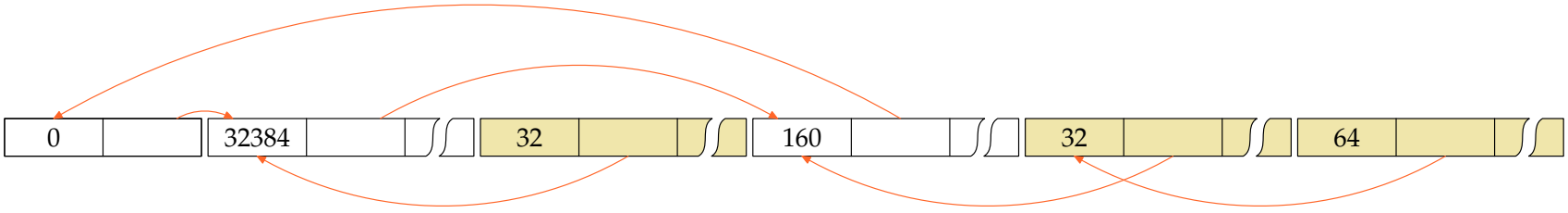


- Wenn nach freien Elementen immer beginnend von dem Spezialelement (*root*) aus gesucht wird, tendiert die Liste der freien Elemente dazu, zu Beginn nur ganz winzige Reste anzubieten, so dass die größeren freien Elemente erst ganz hinten zu finden sind.
- Deswegen wird beim *circular first fit*-Algorithmus die Suche dort fortgesetzt, wo wir zuletzt waren. Und entsprechend dem *first fit* wird die erste Speicherfläche akzeptiert, die genügend Speicherplatz anbietet.
- Das Diagramm zeigt die Situation, wenn 32 Bytes angefordert wurden. Das führte zur Aufspaltung des zuletzt frei gewordenen Elements mit 64 Bytes.

Das Wandern im Ring



- Und wenn erneut Speicher belegt wird, dann beginnt die Suche beim nächsten Element.
- Das ist hier das ganz große freie Element links, von dem wieder etwas am Ende weggenommen wurde.



- Wenn ein Element freigegeben wird, dann kann davor und danach jeweils ein freies Element vorliegen, mit dem das neue Element zusammengelegt werden kann.
- In diesem Beispiel fand sich danach ein freies Element.
- Prinzipiell können bei einer Freigabe bis zu drei Elemente zusammengelegt werden.

Globale Datenstruktur der Speicherverwaltung 244

alloc.c

```
memnode dynmem[MEM_SIZE] = {
   /* bleibt immer im Ring der freien Speicherflaechen */
   {0, &dynmem[1]},
   /* enthaelt zu Beginn den gesamten freien Speicher */
   {sizeof dynmem - 2*sizeof(memnode), dynmem}
};
memnode* node = dynmem;
memnode* root = dynmem;
```

- In dem einfachen Beispiel wird nur Speicher aus dem großen Array *dynmem* vergeben.
- In diesem liegt gleich zu Beginn das Spezialelement, gefolgt von dem großen Element, dem der restliche freie Speicher gehört.
- *root* zeigt immer auf das Spezialelement.
- *node* ist der im Ring herumwandernde Zeiger, der immer auf ein freies Element im Ring verweist.
- Bei einer ernsthaften Implementierung (siehe Übungen und Wettbewerb!) sind dann sukzessive Kacheln vom Betriebssystem zu holen und zu verwalten.

alloc.c

```
memnode* successor(memnode* p) {
   return (memnode*) ((char*)(p+1) + p->size);
}
```

- Das jeweils im Speicher nachfolgende Element zu finden, ist einfach mit Hilfe der Adressarithmetik.
- Zu beachten ist, dass zuerst die Zeigerarithmetik auf Basis des Zeigertyps *memnode*∗ erfolgt mit $p+1$ und dann, um die Größe in Bytes zu addieren, dieser zwischendurch in einen **char**-Zeiger konvertiert werden muss.
- Bei belegten Elementen ist das vorangehende Element immer über den *next*-Zeiger ermittelbar.
- Wenn wir den Ring der freien Elemente durchlaufen, behalten wir immer noch einen Zeiger auf das freie Element davor. Von dem aus können ggf. mit *successor* noch die dazwischenliegenden belegten Speicherelemente durchlaufen werden.
- All diese Tricks stellen sicher, dass die Verwaltungsobjekte nicht zuviel Speicherplatz belegen.

alloc.c

```
void* my_malloc(size_t size) {
   assert(size >= 0);
   if (size == 0) return 0;
   /* runde die gewuenschte Groesse auf
      das naechste Vielfache von ALIGN */
   if (size % ALIGN) {
      size += ALIGN - size % ALIGN;
   }
   /* Suche und Vergabe ... */
}
```

- *malloc* und analog *my_malloc* müssen darauf achten, dass der vergebene Speicher korrekt ausgerichtet ist (Alignment). Am einfachsten ist es hier, alles auf die maximale Alignment-Anforderung auszurichten, das ist hier *ALIGN*.
- Für *ALIGN* wird *alignof*(*max_align_t*) verwendet.

alloc.c

```
memnode* prev = node; memnode* ptr = prev->next;
do {
   if (ptr->size >= size) break; /* passendes Element gefunden */
   prev = ptr; ptr = ptr->next;
} while (ptr != node); /* bis der Ring durchlaufen ist */
if (ptr->size < size) return 0; /* Speicher ist ausgegangen */
```

- *node* ist der im Ring herumwandernde Zeiger auf ein freies Element.
- Wir setzen *prev* auf *node* und *node* gleich auf das nächste Element. Auf diese Weise kennen wir immer den Vorgänger.
- Danach läuft die Schleife, bis entweder ein passendes freies Element gefunden wurde oder wir den gesamten Ring durchlaufen haben.

alloc.c

```
if (ptr->size < size + 2*sizeof(memnode)) {
   node = ptr->next; /* "circular first fit" */
   /* entferne ptr aus dem Ring der freien Speicherflaechen */
   prev->next = ptr->next;
   /* suche nach der unmittelbar vorangehenden Speicherflaeche;
      zu beachten ist hier, dass zwischen prev und ptr noch
      einige belegte Speicherflaechen liegen koennen
   */
   for (memnode* p = prev; p < ptr; p = successor(p)) {
      prev = p;
   }
   ptr->next = prev;
   return (void*) (ptr+1);
}
```

- Wenn das gefundene freie Element genau passt bzw. zu klein ist, um weiter zerlegt zu werden, muss es aus der Liste der freien Element herausgenommen werden.
- Außerdem muss das korrekte Vorgängerelement gefunden werden, auf das der *next*-Zeiger zu verweisen hat.

alloc.c

```
node = ptr; /* "circular first fit" */
/* lege das neue Element an */
memnode* newnode = (memnode*)((char*)ptr + ptr->size - size);
newnode->size = size; newnode->next = ptr;
/* korrigiere den Zeiger der folgenden Speicherflaeche,
   falls sie belegt sein sollte */
memnode* next = successor(ptr);
if (next < dynmem + MEM_SIZE && next->next == ptr) {
   next->next = newnode;
}
/* reduziere die Groesse des alten Elements
   aus dem Ring der freien Speicherflaechen */
ptr->size -= size + sizeof(memnode);
return (void*) (newnode+1);
```

- Andernfalls ist das gefundene freie Element zu zerlegen in ein weiterhin freies Element am Anfang der Fläche und das neue belegte Element.
- Ferner ist darauf zu achten, dass das folgende Element, falls es belegt ist, auf das neugeschaffene Element davor verweist.

alloc.c

```
void my_free(void* ptr) {
   if (!ptr) return;
   memnode* node = (memnode*) ptr - 1;
   assert(is_allocated(node));
   memnode* prev = node->next;
   while (is_allocated(prev)) {
      prev = prev->next;
   }
   node->next = prev->next; prev->next = node;
   join_if_possible(node, node->next);
   join_if_possible(prev, node);
}
```

- Bei belegten Elementen verweist *next* auf das im Speicher zuvor liegende Element, das frei oder belegt sein kann.
- Wir gehen soweit zurück, bis wir das zuvor liegende freie Element finden, damit wir das aktuelle Element in die verkettete Liste der freien Elemente einhängen können.
- Danach wird überprüft, ob ein Zusammenlegen möglich ist.

alloc.c

```
void join_if_possible(memnode* prev, memnode* ptr) {
   memnode* p = (memnode*)((char*) (prev + 1) + prev->size);
   if (p != ptr) return;
   if (prev->size == 0) return;
   prev->next = ptr->next;
   prev->size += ptr->size + sizeof(memnode);
   memnode* next = successor(ptr);
   if (next < dynmem + MEM_SIZE && next->next == ptr) {
      next->next = prev;
   }
   if (node == ptr) node = prev;
}
```

- *prev* und *ptr* verweisen beide auf freie Elemente.
- Wenn *prev* nicht unmittelbar *ptr* vorangeht, lassen sich die beiden Elemente nicht zusammenlegen.
- Wenn zusammengelegt wird, dann ist beim *ptr* folgenden Element der *next*-Zeiger anzupassen, falls dieser zuvor auf *ptr* verwies.

Bei der Speichervergabe kommt es zwangsläufig zu Fragmentierungen, d.h. zum Verlust nutzbarer Speicherfläche. Dabei wird zwischen *interner* und *externer* Fragmentierung unterschieden:

- Interne Fragmentierung entsteht bei der Anwendung des Alignments. Wenn beispielsweise 17 Bytes belegt werden sollen, werden diese bei einer Alignment-Anforderung von 16 auf 32 erhöht, so dass 15 Bytes ungenutzt bleiben. Ebenso zählt die Verwaltungsdatenstruktur (hier 16 Bytes) ebenfalls zur internen Fragmentierung.
- Externe Fragmentierung entsteht, wenn kleine freie Speicherblöcke entstehen, die nicht zu größeren freien Speicherblöcken zusammengelegt werden (können).

Der vorgestellte Algorithmus ist sehr ineffizient:

- Die Zahl der durchzulaufenden freien Elemente ist nur durch die Gesamtzahl der freien Elemente nach oben begrenzt.
- Das Verfolgen einer Zeigerliste ist prinzipiell sehr ineffizient, da die Zugriffe entlang der Zeigerkette sequentiell erfolgen müssen und die folgenden Adressen, von denen geladen wird, nicht von der Hardware „erratbar“ sind (*hardware prefetch*).