#uni/semester3/Betriebssysteme/chapter17/chapter-notes

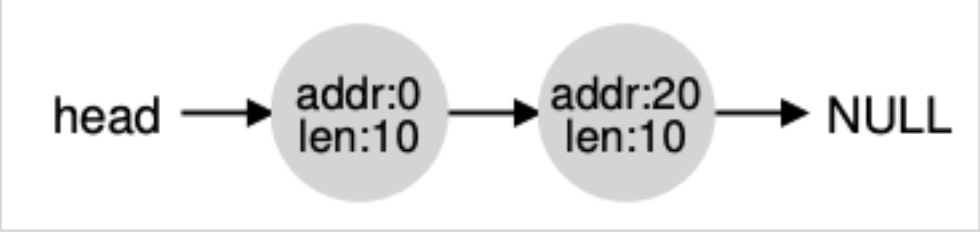
➜ free space management ist einfach, wenn der Bereich ist unterteilt in gleich großen chunks. Es ist komplexer wenn man units mit variablen lengths verwalten muss. Das geschieht zum Beispiel beim malloc oder free. Oder bei OS wenn es physical memory mit segmentation verwaltet.

1. Assumptions
   ➜ malloc returns einen Pointer zu einem Bereich, dass die entsprechende size hat oder größer.
   ➜ free nimmt ein Pointer und gibt ihr Frei. Size muss er dann selber herausfinden
   ➜ Der Bereich dass diese library verwaltet ist bekannt als heap. Die generische Struktur, die die Verwaltung ermöglicht heisst free list. Diese beinhaltet Referenzen zu allen freie Bereiche im heap.
   ➜ internal Fragmentation heisst dass ein Request mehr bekommt als es gefordert hat.
   ➜ Annahme: Speicher dass gegeben wurde (von malloc) kann nicht von der library bewegt werden, bis das Programm es mit free freigegeben hat. Somit ist compaction von freiem Speicher nicht möglich.
   ➜ Annahme: Allocator verwaltet einen contiguous Bereich von Bytes. In manchen cases könnte der allocator, nach mehr Speicher fürs heap fordern (mit sbrk zum Beispiel). Aber hier wird angenommen, dass ein Bereich eine fixed size solange er lebt, hat.

2. Low-level Mechanisms

### 2.1 Splitting and Coalescing


➜ die free list würde dann zwei Elemente haben.




➜ ein Request, für eine size von größer 10, würde dann NULL bekommen.

➜ Wenn ein Request eine size kleiner als 10 anfordert, dann macht der allocator splitting. Das heisst er sucht nach einem Bereich, dass den request erfüllt, und unterteilt ihn in zwei Bereiche. Der erste chunk bekommt der caller. Der zweite chunk bleibt auf der free list.
➜ Wenn man ein Speicherbereich freigibt, dann schaut der allocator ob der Bereich Nachbaren hat, die auch frei sind. Wenn ja, dann wird alles zu einem verschmelzt. Das nennt man coalescing. Dadurch versichert der allocator, dass larger memory chunks für das Programm zur Verfügung stehen.

**2.2 Tracking The Size Of Allocated Regions**

➜ free hat kein size parameter, d. h. Die malloc library kann dann die Größe der Bereichs determinieren. Um das zu ermöglichen, der allocator speichert extra Informationen in einem header block, vor dem allokierten Bereich.
➜ der header beinhaltet die size des allokierten Bereichs, zusätzliche Pointer um die Deallokierung schneller zu machen, eine magic number um integrity checking zu ermöglichen und andere Informationen (siehe chapter 14 notizen)
➜ Wenn user ruft free mit pointer, dann die library berechnet mittels Pointerarithmetik, wo der header beginnt.

```
void free(void *ptr) {
    header_t *hptr = (header_t *) ptr - 1;
    ...
```

➜ Dann wird die magic number überprüft.

```
(assert(hptr->magic == 1234567))
```

➜ Dann wird die size des ganzen Bereichs berechnet (size von header + size von allokierten Bereich). Deshalb wenn user N bytes fordert, dann sucht die library nach einem chunk von Größe N + size of header.

**2.3 Embedding A Free List**

➜ Die free list muss in dem freien Bereich aufgebaut werden.
➜ Beispiel, wo wir 4096 Bytes of memory verwalten. Um das als eine free list zu verwalten, muss zuerst die list initialisiert werden. Am Anfang hat sie nur ein entry, von size 4096 - header size.
➜ Ein node in der Liste sieht so aus:

```
typedef struct __node_t {
    int              size;
    struct __node_t *next;
} node_t;
```

➜ heap wird in einem freien Bereich aufgebaut, dass von dem systemcall mmap zurückgeliefert wird.

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size   = 4096 - sizeof(node_t);
head->next   = NULL;
```

➜ Der head pointer beinhaltet dann die Startadresse dieses Bereichs.

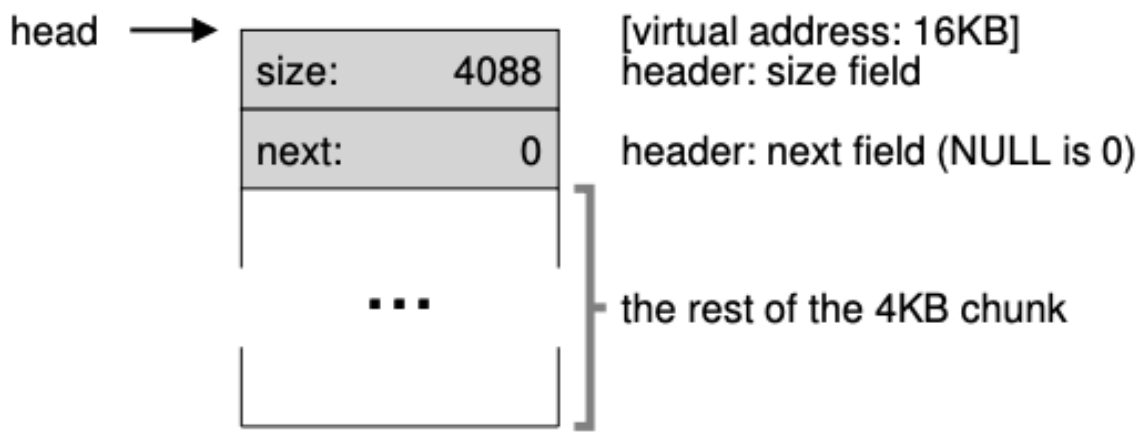



Figure 17.3: **A Heap With One Free Chunk**

➜ Wenn e.g. 100 Bytes gefordert werden, findet die library einen chunk der large genug ist (In dem Beispiel haben wir nur ein chunk). Dieser wird dann gesplittet, in 100 Bytes + header size und der restliche freier Bereich. Header kann zum Beispiel die size von 8 Byte haben (eine integer size und integer magic number). Somit hat die library 108 Bytes allokiert und einen Pointer returned.

➜ Wenn man ein free auf ein chunk macht dann wird auf diesem der head pointer gelegt. Und beinhaltet danach noch im header ein next pointer zu dem nächsten freien Bereich.

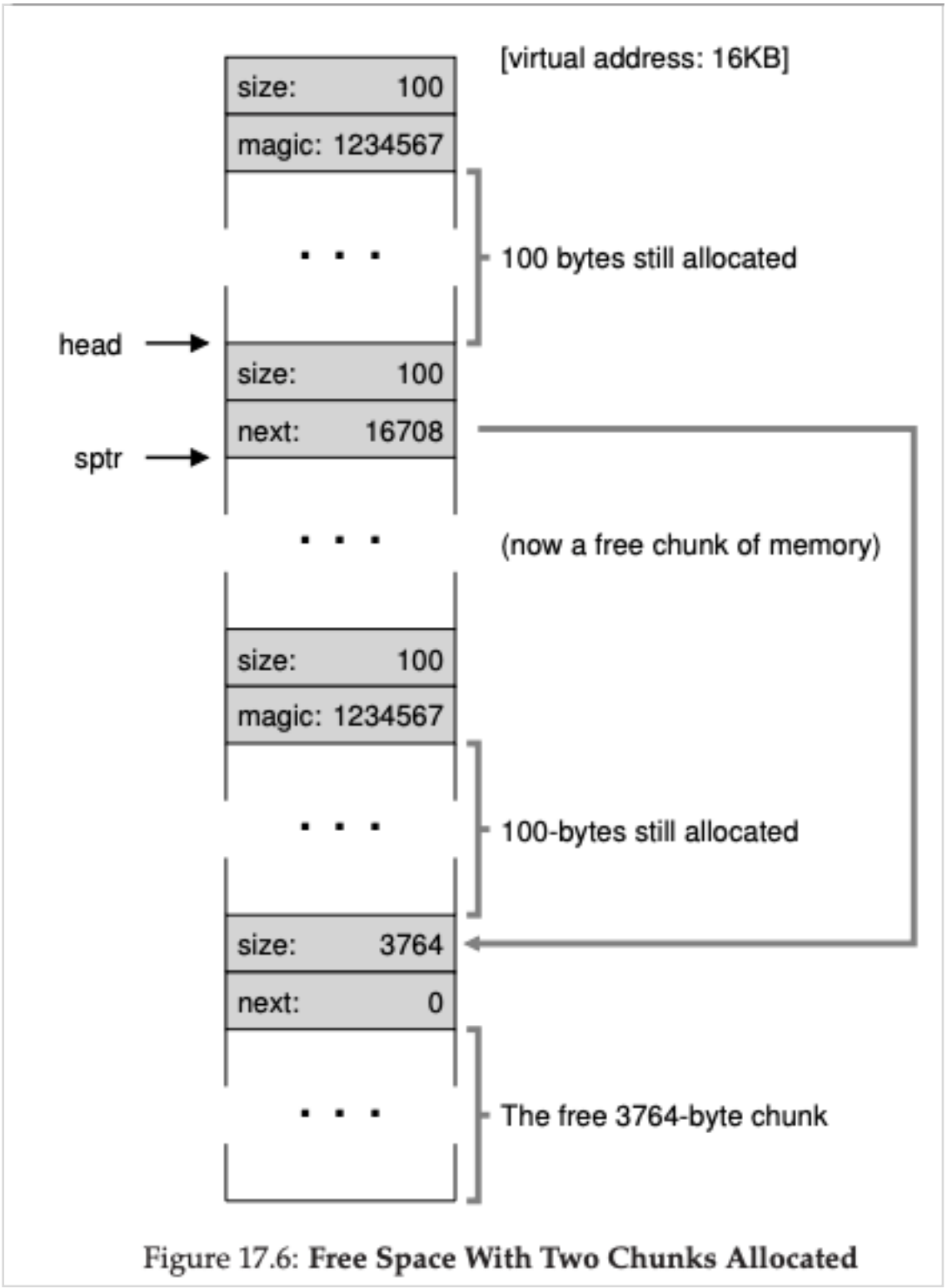



Figure 17.6: **Free Space With Two Chunks Allocated**

**2.4 Growing The Heap**

➜ Was sollte passieren wenn der heap kein freier Speicher mehr hat? NULL zurückzulieferen in bestimmten Fälle ist in Ordnung

➜ Die meisten allocators, starten mit einem kleinen heap und während der Laufzeit fordern dann mehr. Das passiert über ein system call wie sbrk um heap zu vergrößern. Dann der allocator selber verwaltet dieses neuen Bereich.

➜ Um den sbrk request zu erfüllen, findet das OS freie physical pages und mappt sie auf

dem Adressbereich des Prozesses. Am Schluss liefert das OS dann das Ende des neuen heaps. (Lies mal sbrk manual)

3. Basic Strategies

➜ der ideale allocator ist schnell und minimiert fragmentation. Es gibt leider kein best approach.
➜ Best Fit sucht in der free list nach chunks die gleich gross oder grösser sind als der request. Es wird dann der genommen, der am nahesten ist zu was der Prozess möchte. Vorteil, es reduziert verschwendeter Speicher. Nachteil, die das suchen ist ein grosser overhead für die Leistung.
➜ Worst fit ist Gegenteil von Best fit. Er versucht grosse free chunks Hinterzulassen, anstatt viele von kleinere chunks. Nachteil, suchen ist wieder ein grosser overhead und verursacht excessive fragmentation
➜ First fit nimmt das erste gross genug free chunk. Es ist schnell, aber manchmal führt dazu, dass der Anfang der free list kleine Objekte enthält. Dadurch entsteht eine neues Problem nämlich wie der allocator die Ordnung der free list verwaltet (das führt dazu vielleicht dass first fit dadurch mehr in die list rein suchen muss). Ein approach ist eine address-based ordering zu verwenden. Somit wird die Liste nach der Adresse des free space sortiert. Coalescing wird einfacher und fragmentation reduziert.
➜ Next fit arbeitet wie first fit, aber nach einem Suchen wird ein Pointer zurückgelassen. Hier wird dann beim nächsten mal weiter gemacht. Die Suchen werden dann uniform auf der ganzen Liste gelassen.

4. Other Approaches

**4.1 Segregated Lists**

➜ Basic idea is if an application has a popular-sized request, keep a separate list just to manage objects of that size. The rest would be directed to a more general memory allocator.
➜ Advantages are, that fragmentation would be a lesser problem. Moreover allocation and free requests will be much faster, because there won't have to be a complicated search
➜ New problem, how much memory should you give to a such specialised list as opposed to the general pool?
➜ slab allocator handles this in a nice way. When the kernel boots up, a number of object caches for kernel objects are allocated. These object caches are segregated lists. When a

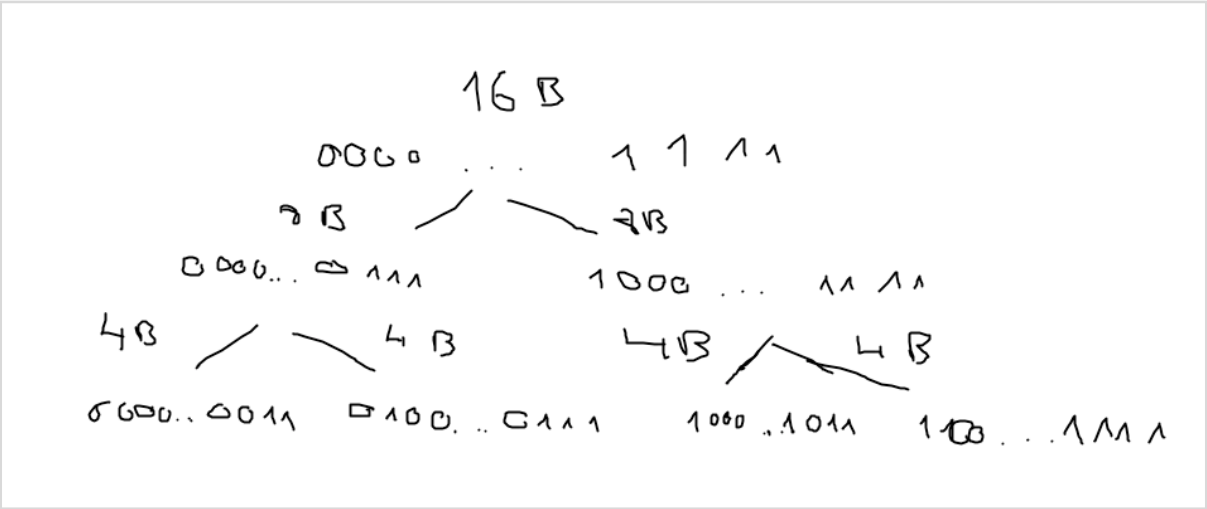
cache runs low on memory, it requests some slabs from the general memory allocator. When the count of objects in this slab is zero (or vm needs more memory), the general allocator can reclaim them black. Moreover it keeps the free objects on the list in a pre-initiliazed state. This way frequent initialisation and destruction cycles are avoided and thus lower overheads.

### 4.2 Buddy Allocation

➜ Binary buddy allocator helps a lot with coalescing, by making it easy

➜ Free memory is viewed as a big space of 2^N. When a request is made, the free space is divided by two until a block that is big enough is found and the next split wouldn't be enough. Then the left most block is allocated. This might lead to internal fragmentation. When the block is freed, it checks if the other buddy is free. If so, it then coalesces the two blocks into one. This recursive process continues up the tree.

➜ It is simple to the determine the buddy of a particular block. The address of each buddy pair one differs by a single bit, which bit is determined by the level in the buddy tree.




### 4.3 Other ideas

➜ One problem with the described approaches is they scaling. Searching lists can be slow, thus advanced allocators use more complex data structures to address these costs, by trading simplicity for performance. (Binary trees, splay trees, etc..)

➜ Modern allocators are designed to work well on multiprocessor based systems.

➜ Read about the glibc allcator!

➜ YouTube slab allocator