

Heap og concurrency

Processes, Threads og races

- Proces
 - Program der kører med egen "address space"
- Threads
 - Kører i samme address space som processen der spawned den
 - Flere threads kan eksisterer i en process, deler derfor ressourcer
- Race conditions
 - Opstår når flere threads tilgår eller ændrer fælles data ("critical section")
 - Non-deterministic, resultat afhænger af timing
 - Kan fikses ved brug af mutex locks
- Deadlock
 - Opstår når to eller flere threads blokerer hinanden permanent fordi de venter på ressourcer holdt af andre threads

Concurrent og parallel

- Parallel: To eller flere logical flows på præcis samme tid
- Concurrent: To eller flere logical flows der kan ske i hvilken som helst rækkefølge

Mutexes

- Bruges til at beskytte "critical sections"
 - Kode som udelukkende en thread må udfører ad gangen

```
// shared mutex instance  
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_lock(&mutex);  
... critical section ...  
pthread_mutex_unlock(&mutex);
```

Andet

- Threadpools
 - Kollektion af "worker threads"
 - Udfører den givne opgave og venter derefter på en anden opgave
 - Billigere end at starte og ødelægge threads en masse threads
- Condition variables
 - Simpel værdi der bruges til at kommunikerer til andre processer

Question 2.3.2: Consider the following C code which increments a variable count by some amount n

```
1  for (int i = 0; i < n; i++) {  
2      count++;  
3  }
```

Now consider the following Assembly code for incrementing a value for count in a register. It has 3 labelled sections, L to load the value of count, U to update the value of count, and S to save the value of count.

```
        li t0, 0           # i = 0  
        la t1, count       # address of count in t1  
        beq t0, a0, done   # skip if nothing to do  
  
loop:  
    L { lw t2, 0(t1)       # load count from memory  
      U { addi t2, t2, 1    # increment count  
      S { sw t2, 0(t1)     # store count in memory  
        addi t0, t0, 1     # i++  
        beq t0, a0, done   # done?  
        j loop            # another iteration  
  
done:
```

We could express the order of operations for a single thread running the above code with $n = 2$ like so:

$L_0 U_0 S_0 L_0 U_0 S_0$

E.g. Thread 0 loads, updates and saves twice, once for each iteration of the loop. We could express two identical threads running the above code with $n = 1$ as:

$L_0 U_0 S_0 L_1 U_1 S_1$

E.g. Thread 0 loads, updates and saves once, and then Thread 1 loads, updates and saves once. Both of the above would produce a final value for count of 2, as no race condition has occurred.

What would be a potential ordering of operations for two threads and $n=1$, where a race condition occurs and the final value for count is 1? Assume count starts set to 0.

What would be a potential ordering of operations for two threads and $n=2$, where a race condition occurs and the final value for count is 3? Assume count starts set to 0.

What would be a potential ordering of operations for two threads and $n=3$, where a race condition occurs and the final value for count is 2? Assume count starts set to 0.

```
        li t0, 0          # i = 0
        la t1, count      # address of count in t1
        beq t0, a0, done  # skip if nothing to do
loop:
    L { lw t2, 0(t1)      # load count from memory
      U { addi t2, t2, 1   # increment count
      S { sw t2, 0(t1)    # store count in memory
        addi t0, t0, 1    # i++
        beq t0, a0, done  # done?
        j loop            # another iteration
done:
```

What would be a potential ordering of operations for two threads and $n=1$, where a race condition occurs and the final value for count is 1? Assume count starts set to 0.

$L_0 L_1 U_0 S_0 U_1 S_1$

What would be a potential ordering of operations for two threads and $n=2$, where a race condition occurs and the final value for count is 3? Assume count starts set to 0.

$L_0 L_1 U_0 S_0 L_0 U_0 S_0 U_1 S_1 L_1 U_1 S_1$

What would be a potential ordering of operations for two threads and $n=3$, where a race condition occurs and the final value for count is 2? Assume count starts set to 0.

$L_0 U_0 L_1 U_1 S_1 L_1 U_1 S_1 S_0 L_1 U_1 L_0 U_0 S_0 L_0 U_0 S_0 S_1$

```

                                li t0, 0           # i = 0
                                la t1, count         # address of count in t1
                                beq t0, a0, done      # skip if nothing to do

loop:
    L { lw t2, 0(t1)           # load count from memory
      U { addi t2, t2, 1        # increment count
      S { sw t2, 0(t1)         # store count in memory
        addi t0, t0, 1         # i++
        beq t0, a0, done      # done?
        j loop                # another iteration

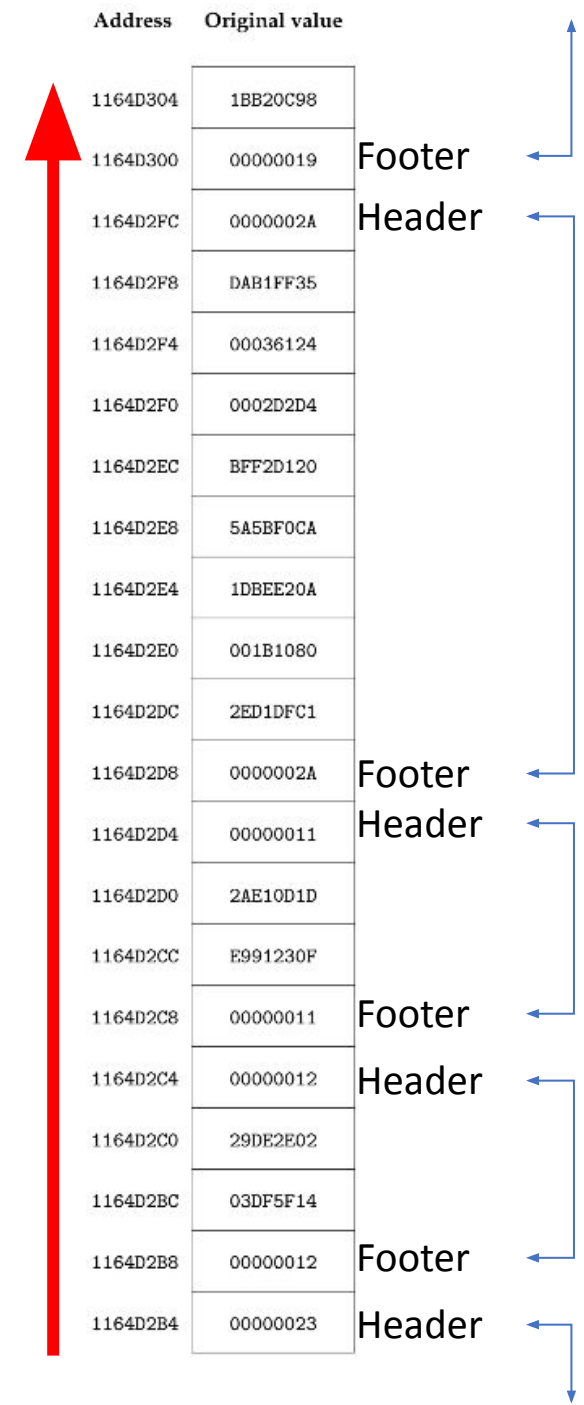
done:
```

Nu til heapen

Basis

- Heapen gror fra bunden og op
- Består af blocks indikeret af headers og footers
 - Headers og footers indeholder information vedrørende blocken indkodet i bitsne
 - Headeren og footeren for en block har altid samme værdi
 - Headeren og footeren er 4 bytes hver og tæller med i block størrelsen
- Hver adresse er 4 bytes
- En block størrelse skal være multipel af 8

Address	Original value	
1164D304	1BB20C98	
1164D300	00000019	Footer
1164D2FC	0000002A	Header
1164D2F8	DAB1FF35	
1164D2F4	00036124	
1164D2F0	0002D2D4	
1164D2EC	BFF2D120	
1164D2E8	5A5BF0CA	
1164D2E4	1DBEE20A	
1164D2E0	001B1080	
1164D2DC	2ED1DFC1	
1164D2D8	0000002A	Footer
1164D2D4	00000011	Header
1164D2D0	2AE10D1D	
1164D2CC	E991230F	
1164D2C8	00000011	Footer
1164D2C4	00000012	Header
1164D2C0	29DE2E02	
1164D2BC	03DF5F14	
1164D2B8	00000012	Footer
1164D2B4	00000023	Header



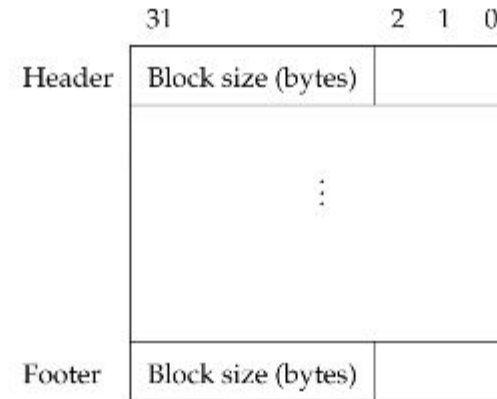
Afkodning af værdierne

- Binære repræsentation
 - Bit 0 – Blocken er fri eller allokeret
 - Bit 1 – Forrige block er fri eller allokeret
 - Bit 2 – Altid sat til 0
- 0x12 -> 00010**010**
- Størrelse findes ved at sætte bitsne til 0
 - 00010**000** = 16

Eksamen fra 24-25

- Husk at læs opgaven!

Question 2.1.3: Consider a memory allocator that uses an implicit free list and immediate coalescing of neighbouring free blocks. The layout of each allocated and free memory block is as follows, with one 32-bit word per row:



Each memory block, either allocated or free, has a size in bytes, rounding up allocations if necessary so that each block is a multiple of 8 bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. Blocks of size 8 are not allowed (so the minimum size of a block is 16 bytes). The usage of the remaining 3 lower order bits is as follows:

- Bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- Bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- Bit 2 is unused and is always set to be 0.

Given the partial contents of the heap shown on the left, show the new contents of the heap after a call `ptr = malloc(12)` is executed (in the middle-left column), followed by a call `free(1164D2CC)` (middle-right column), and finally after a call of `free(ptr)` (rightmost column)

- All addresses and values in memory are hexadecimal, as should be your answers.
- Note that the address grows from bottom up.
- Some parts of the heap may lie outside the area shown.
- Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.
- Perform the minimum number of memory changes required.

- ptr = malloc(12)
- free(1164D2CC)
- free(ptr)

Address	Original value	After		
		ptr = malloc(12)	free(1164D2CC)	free(ptr)
1164D304	1BB20C98	1BB20C98	1BB20C98	1BB20C98
1164D300	00000019	00000019	00000019	00000019
1164D2FC	0000002A			
1164D2F8	DAB1FF35	DAB1FF35	DAB1FF35	DAB1FF35
1164D2F4	00036124			
1164D2F0	0002D2D4			
1164D2EC	BFF2D120			
1164D2E8	5A5BFOCA	5A5BFOCA	5A5BFOCA	5A5BFOCA
1164D2E4	1DBEE20A	1DBEE20A	1DBEE20A	1DBEE20A
1164D2E0	001B1080	001B1080		
1164D2DC	2ED1DFC1	2ED1DFC1		
1164D2D8	0000002A			
1164D2D4	00000011			
1164D2D0	2AE10D1D			
1164D2CC	E991230F			
1164D2C8	00000011	00000011		
1164D2C4	00000012	00000012		
1164D2C0	29DE2E02	29DE2E02	29DE2E02	29DE2E02
1164D2BC	03DF5F14			
1164D2B8	00000012			
1164D2B4	00000023	00000023	00000023	00000023

- Start med at finde headers og footers
 - 0x23 -> 00100**011**
 - Blocken er allokeret
 - Forrige block er allokeret
 - Størrelse = 00100**000** = 32

Address	Original value	
1164D304	1BB20C98	
1164D300	00000019	Footer
1164D2FC	0000002A	Header
1164D2F8	DAB1FF35	
1164D2F4	00036124	
1164D2F0	0002D2D4	
1164D2EC	BFF2D120	
1164D2E8	5A5BF0CA	
1164D2E4	1DBEE20A	
1164D2E0	001B1080	
1164D2DC	2ED1DFC1	
1164D2D8	0000002A	Footer
1164D2D4	00000011	Header
1164D2D0	2AE10D1D	
1164D2CC	E991230F	
1164D2C8	00000011	Footer
1164D2C4	00000012	Header
1164D2C0	29DE2E02	
1164D2BC	03DF5F14	
1164D2B8	00000012	Footer
1164D2B4	00000023	Header

32 (8)

Eksamen fra 24-25

- Start med at finde headers og footers
 - 0x23 -> 00100**011**
 - Blocken er allokeret
 - Forrige block er allokeret
 - Størrelse = 00100**000** = 32
 - 0x12 -> 00010**010**
 - Blocken er fri
 - Forrige block er allokeret
 - Størrelse = 00010**000** = 16
 - Osv...

Address	Original value	
1164D304	1BB20C98	
1164D300	00000019	Footer
1164D2FC	0000002A	Header
1164D2F8	DAB1FF35	
1164D2F4	00036124	
1164D2F0	0002D2D4	
1164D2EC	BFF2D120	
1164D2E8	5A5BF0CA	
1164D2E4	1DBEE20A	
1164D2E0	001B1080	
1164D2DC	2ED1DFC1	
1164D2D8	0000002A	Footer
1164D2D4	00000011	Header
1164D2D0	2AE10D1D	
1164D2CC	E991230F	
1164D2C8	00000011	Footer
1164D2C4	00000012	Header
1164D2C0	29DE2E02	
1164D2BC	03DF5F14	
1164D2B8	00000012	Footer
1164D2B4	00000023	Header

16 (4)

32 (8)

- Start med at finde headers og footers

- 0x23 -> 00100**011**
 - Blocken er allokeret
 - Forrige block er allokeret
 - Størrelse = 00100**000** = 32
- 0x12 -> 00010**010**
 - Blocken er fri
 - Forrige block er allokeret
 - Størrelse = 00010**000** = 16
- Osv...

Address	Original value		
1164D304	1BB20C98		
1164D300	00000019	Footer	24 (6)
1164D2FC	0000002A	Header	
1164D2F8	DAB1FF35		
1164D2F4	00036124		
1164D2F0	0002D2D4		
1164D2EC	BFF2D120		
1164D2E8	5A5BF0CA		
1164D2E4	1DBEE20A		
1164D2E0	001B1080		
1164D2DC	2ED1DFC1		
1164D2D8	0000002A	Footer	40 (10)
1164D2D4	00000011	Header	
1164D2D0	2AE10D1D		
1164D2CC	E991230F		
1164D2C8	00000011	Footer	16 (4)
1164D2C4	00000012	Header	
1164D2C0	29DE2E02		
1164D2BC	03DF5F14		
1164D2B8	00000012	Footer	16 (4)
1164D2B4	00000023	Header	32 (8)

- `ptr = malloc(12)`
 - Allokerer 12 bytes
- Opdater header og footer i den nye allokerede block
 - Ny størrelse = 12 bytes + header og footer
 - 24 = 00011000 (multiple af 8)
- Opdater bitsne
 - Blocken er allokeret
 - Forrige er allokeret
 - 00011011 = 0x1B

Address	Original value		
1164D304	1BB20C98		
1164D300	00000019	Footer	24 (6)
1164D2FC	0000002A	Header	
1164D2F8	DAB1FF35		
1164D2F4	00036124		
1164D2F0	0002D2D4		
1164D2EC	BFF2D120		
1164D2E8	5A5BF0CA		
1164D2E4	1DBEE20A		
1164D2E0	001B1080		
1164D2DC	2ED1DFC1		
1164D2D8	0000002A	Footer	40 (10)
1164D2D4	00000011	Header	
1164D2D0	2AE10D1D		
1164D2CC	E991230F		
1164D2C8	00000011	Footer	16 (4)
1164D2C4	00000012	Header	
1164D2C0	29DE2E02		
1164D2BC	03DF5F14		
1164D2B8	00000012	Footer	16 (4)
1164D2B4	00000023	Header	32 (8)

Address	Original value	
1164D304	1BB20C98	
1164D300	00000019	Footer
1164D2FC	0000002A	Header
1164D2F8	DAB1FF35	
1164D2F4	00036124	
1164D2F0	0002D2D4	
1164D2EC	BFF2D120	
1164D2E8	5A5BF0CA	
1164D2E4	1DBEE20A	
1164D2E0	001B1080	
1164D2DC	2ED1DFC1	
1164D2D8	0000002A	Footer
1164D2D4	00000011	Header
1164D2D0	2AE10D1D	
1164D2CC	E991230F	
1164D2C8	00000011	Footer
1164D2C4	00000012	Header
1164D2C0	29DE2E02	
1164D2BC	03DF5F14	
1164D2B8	00000012	Footer
1164D2B4	00000023	Header

24 (6)

40 (10)

16 (4)

16 (4)

32 (8)

After		
ptr = malloc(12)		
1164D304	1BB20C98	
1164D300	00000019	
1164D2FC		
1164D2F8	DAB1FF35	
1164D2F4		
1164D2F0		
1164D2EC	0000001B	Header
1164D2E8	5A5BF0CA	
1164D2E4	1DBEE20A	
1164D2E0	001B1080	
1164D2DC	2ED1DFC1	
1164D2D8	0000001B	Footer
1164D2D4	00000011	Header
1164D2D0	2AE10D1D	
1164D2CC	E991230F	
1164D2C8	00000011	Footer
1164D2C4	00000012	Header
1164D2C0	29DE2E02	
1164D2BC	03DF5F14	
1164D2B8	00000012	Footer
1164D2B4	00000023	Header

24 (6)

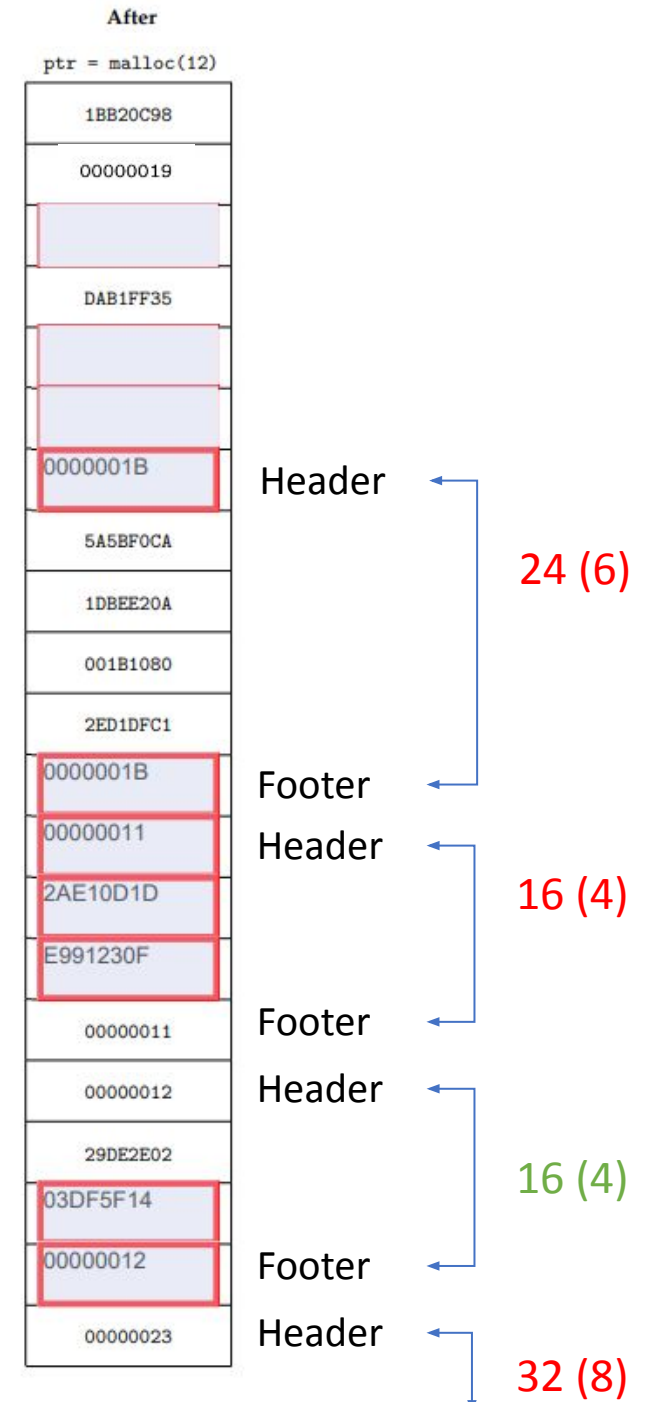
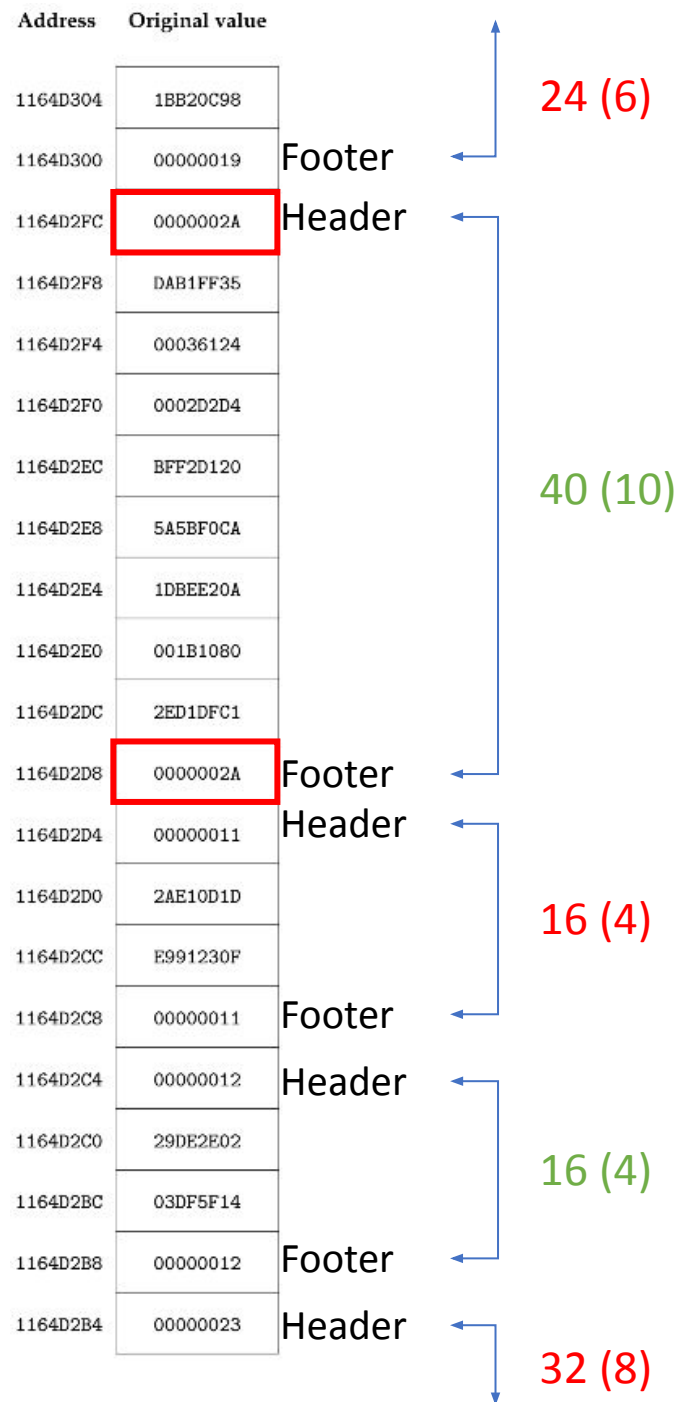
16 (4)

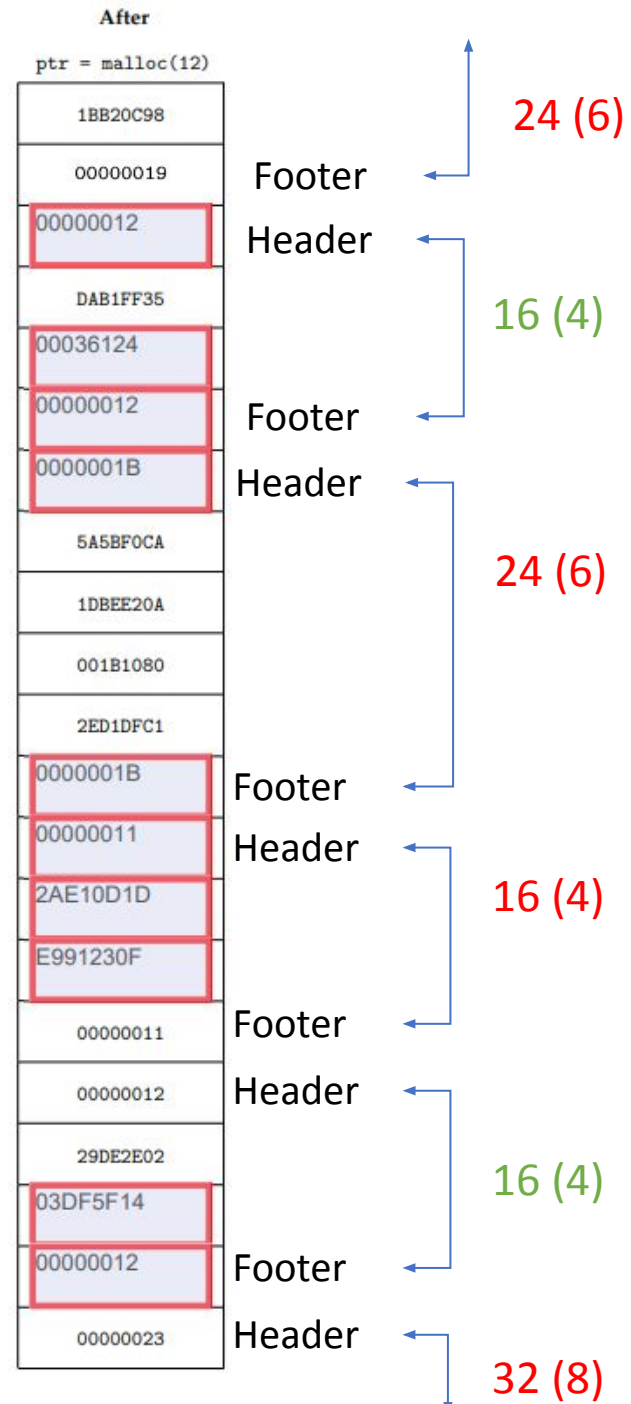
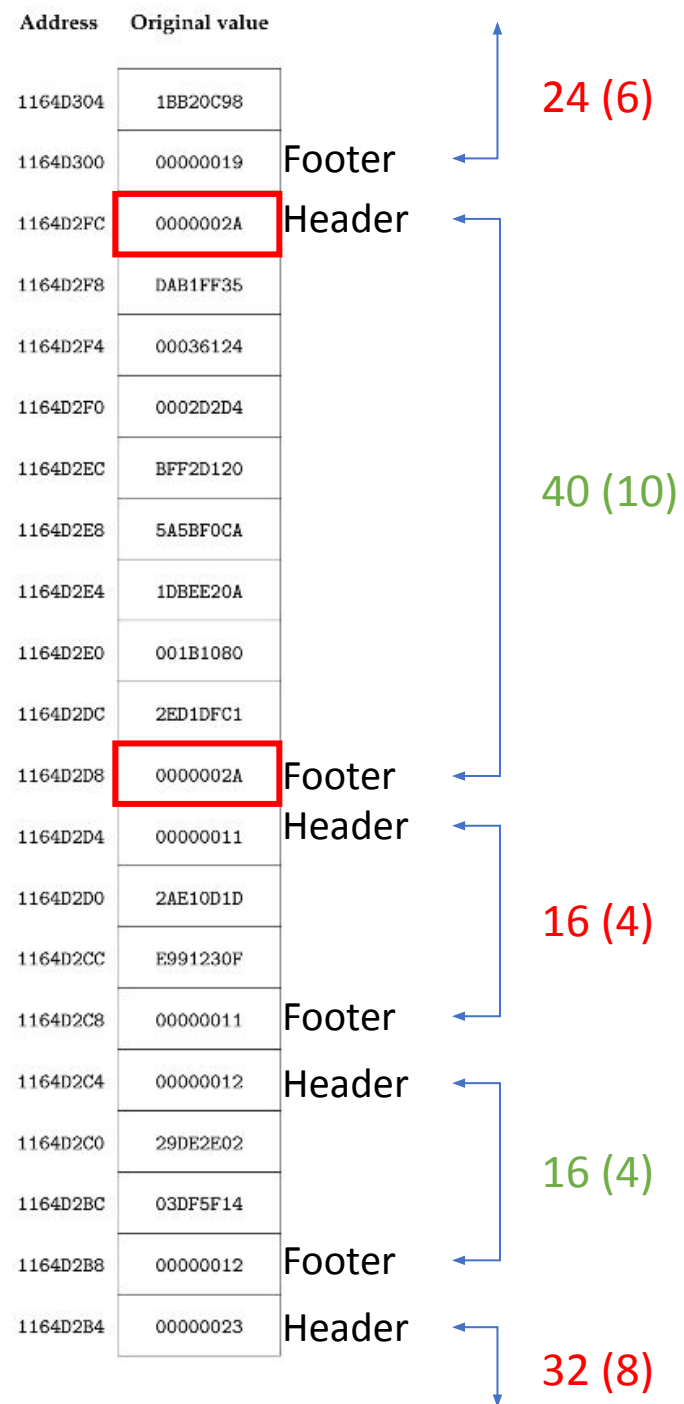
16 (4)

32 (8)

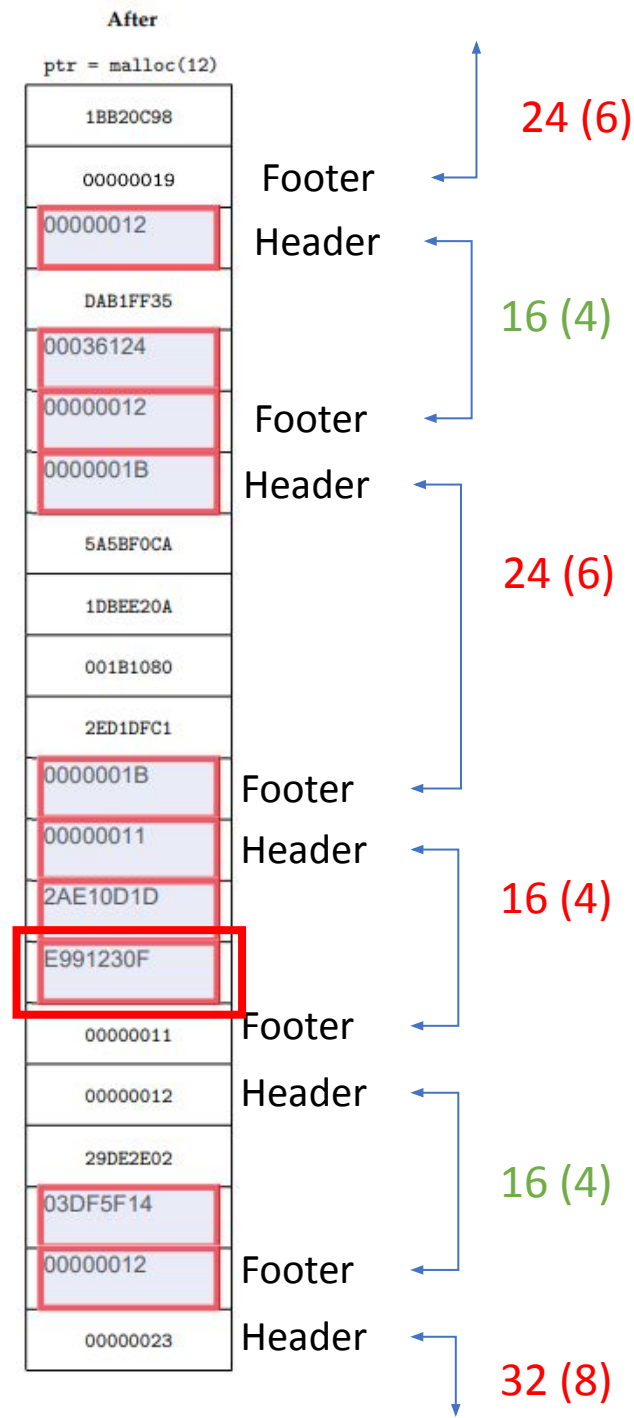
- Resten af den tidligere frie block skal have en header og footer

- Blocken er fri
- Tidligere block er allokeret
- Størrelsen = 16 = 00010**000**
- 00010**010** = 0x12

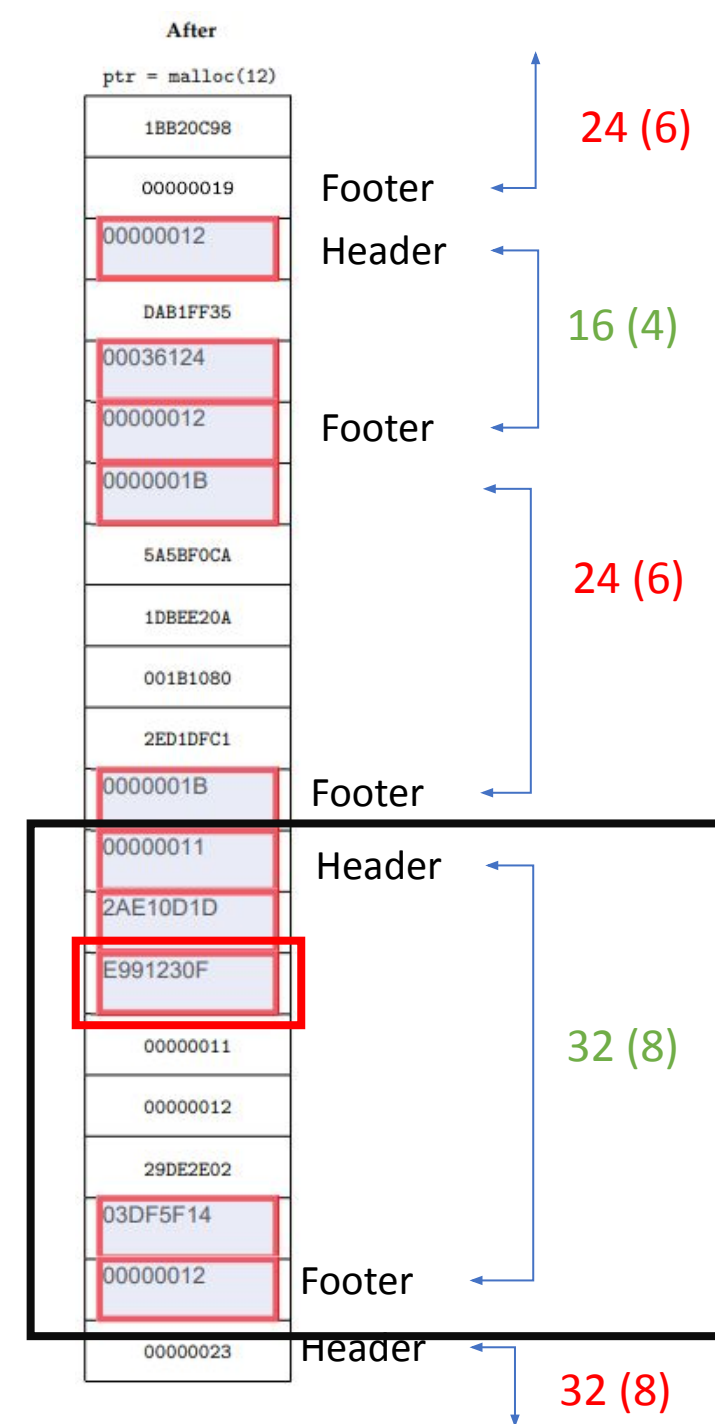
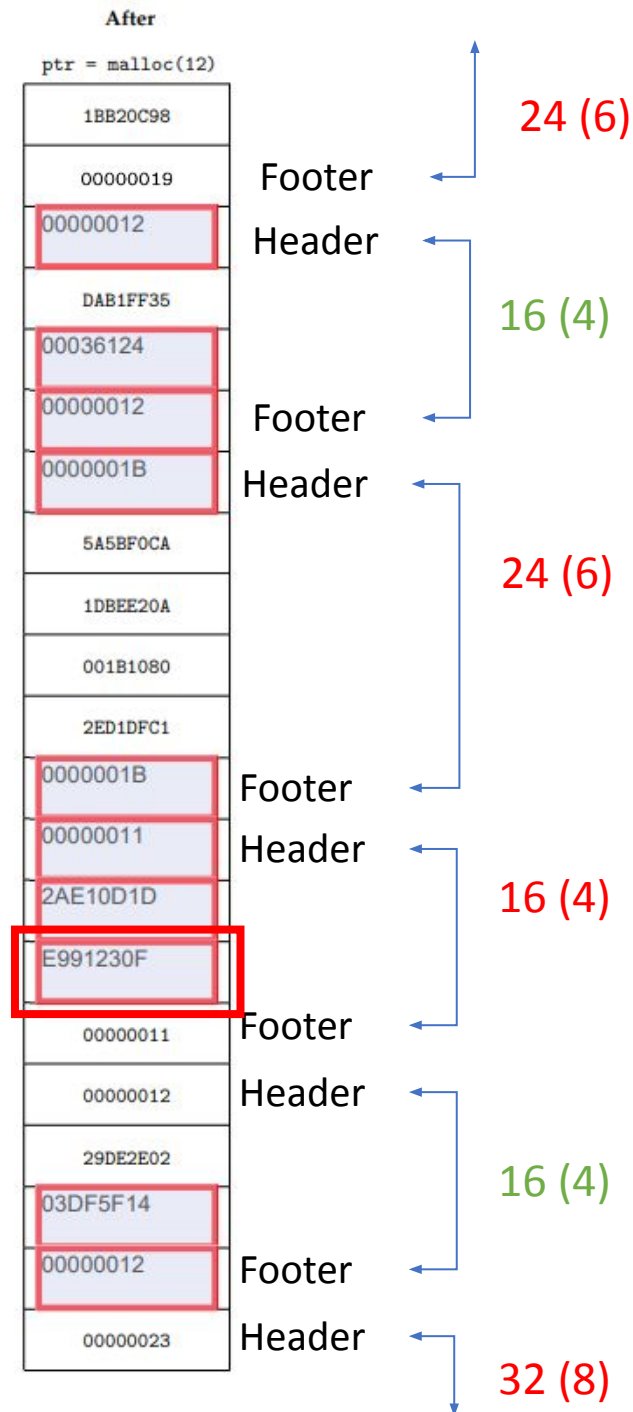




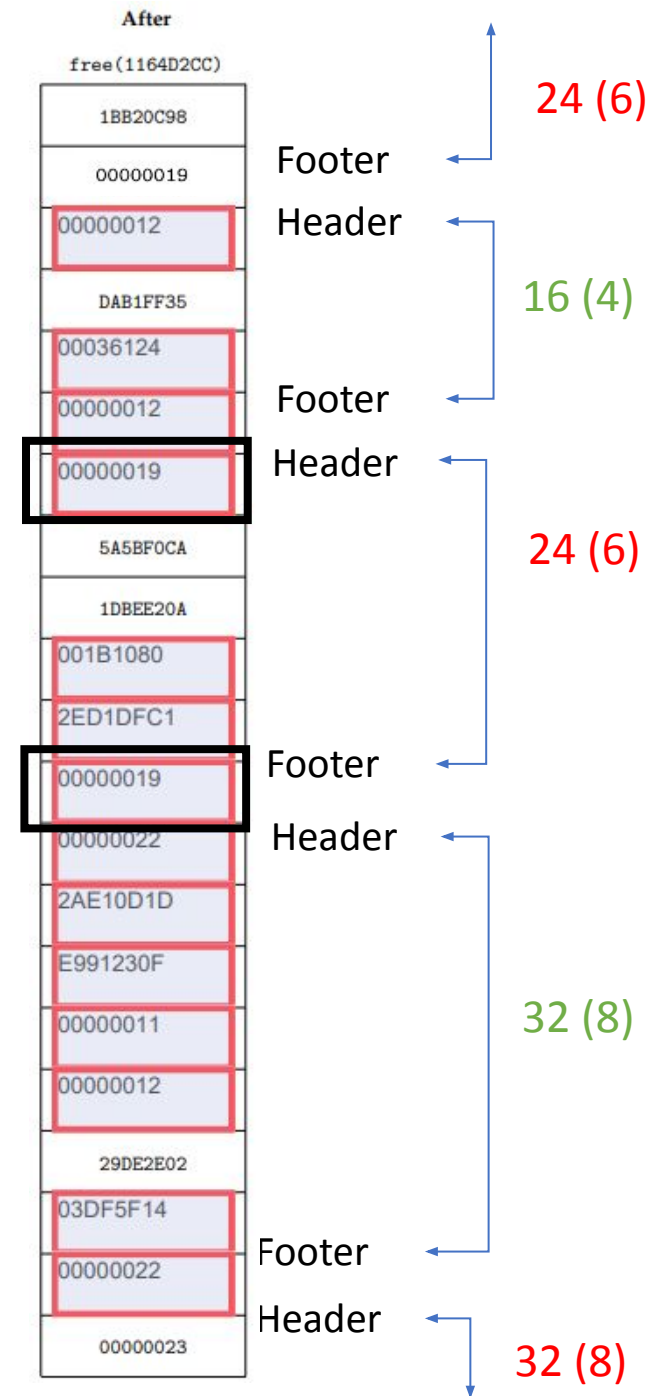
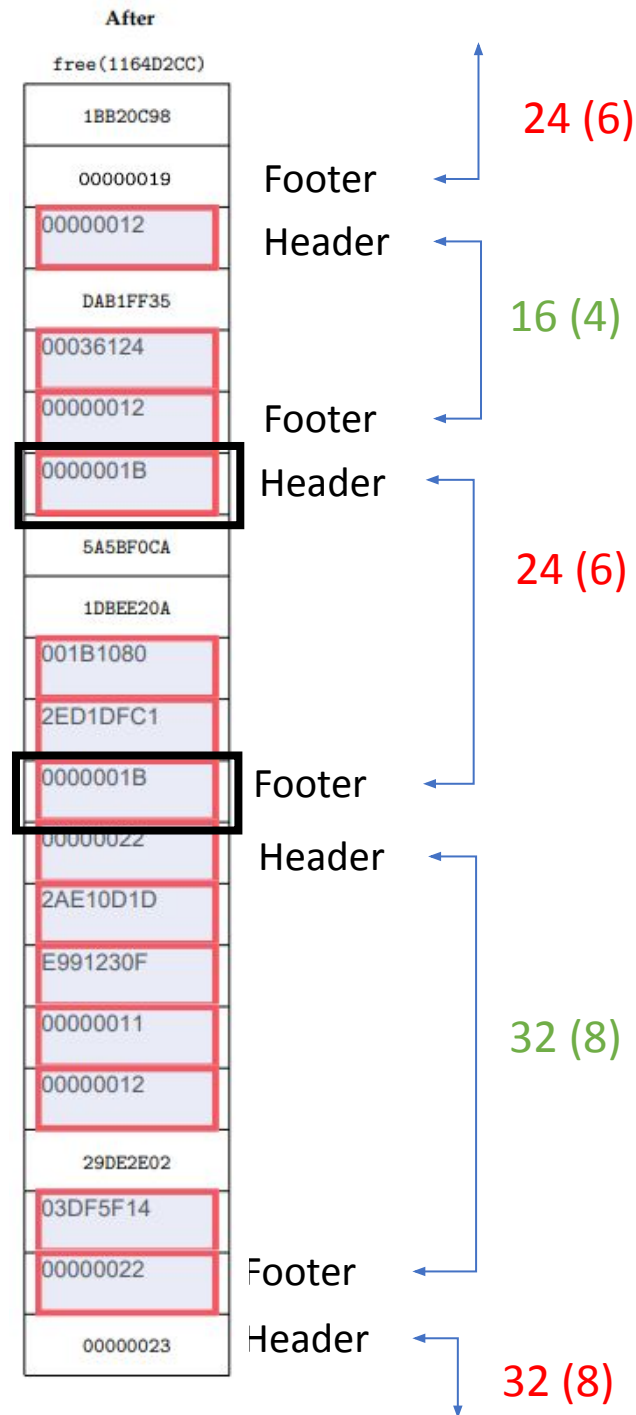
- free(1164D2CC)
- Set bit 0 i header og footer til 0, for at fri den
- Check for "immediate coalescing"



- Der kommer 2 frie blocks ved siden af hinanden = immediate coalescing
- Opdater header og footer
 - Blocken er fri
 - Tidligere block er allokeret
 - Størrelsen = 32 = 00100000
 - 00100010 = 0x22



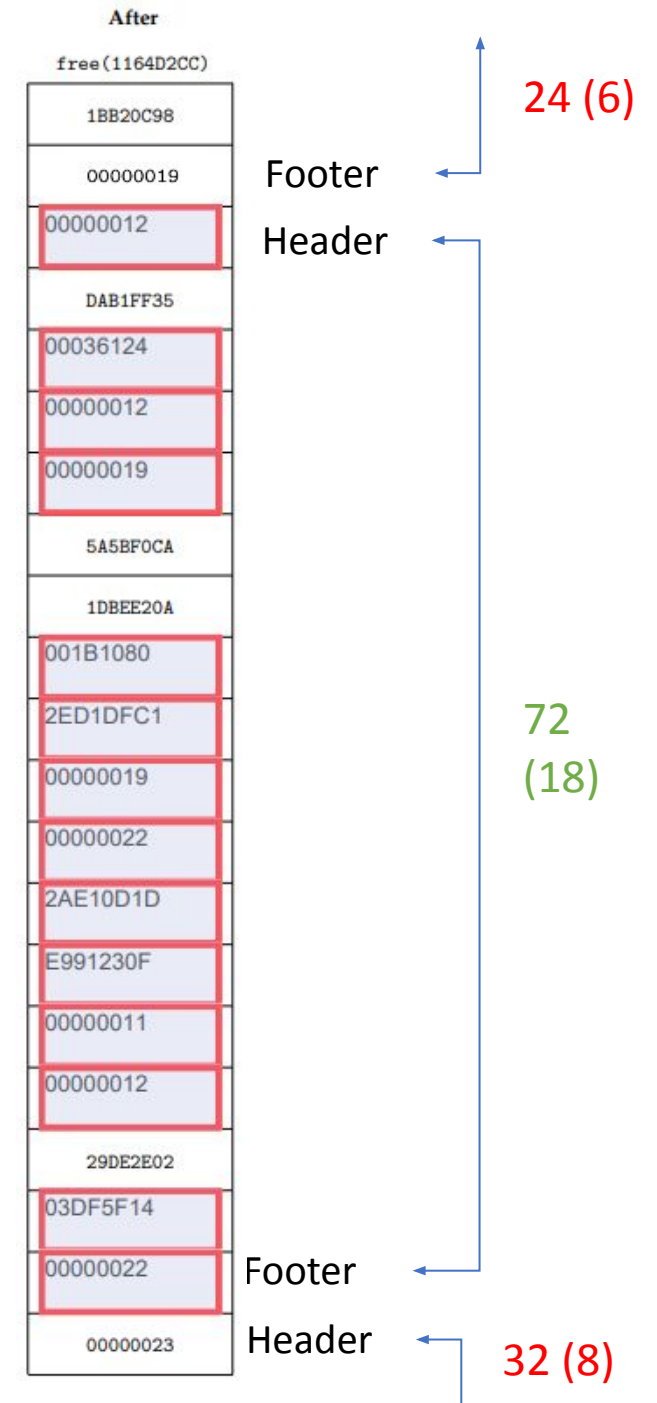
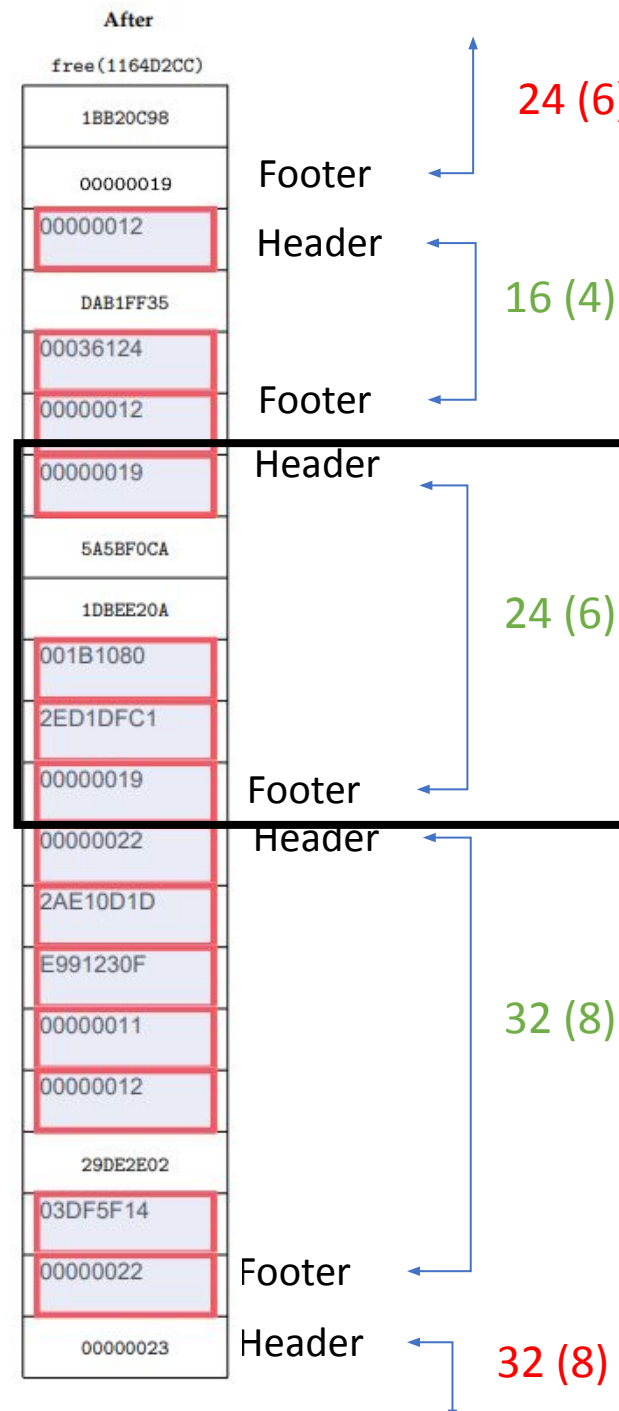
- Blocken efter er nu forket!
- Opdater header og footer (0x1B) da forrige block nu er fri
- Sæt bit 1 til 0
- 00011001 = 0x19



- free(ptr)
 - ptr = malloc(12)
 - Malloc returnere en pointer til den første byte i den allokerede blok
 - ptr = 1164D2D8
- Check for "immediate coalescing"



- Der kommer 3 frie blocks ved siden af hinanden
- Opdater header og footer
 - Blocken er fri
 - Tidligere block er allokeret
 - Størrelsen = 72 = 01001000
 - 00100010 = 0x4A



- Blocken efter er allerede rigtig
 - $0x19 = 00011001$
 - Bit 1 viser at forrige ikke er allokeret hvilket er rigtigt
- Der skal ikke gøres mere

