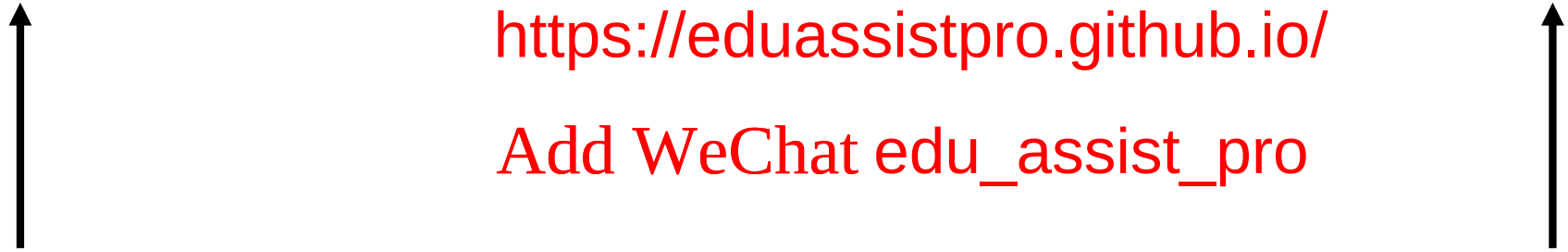
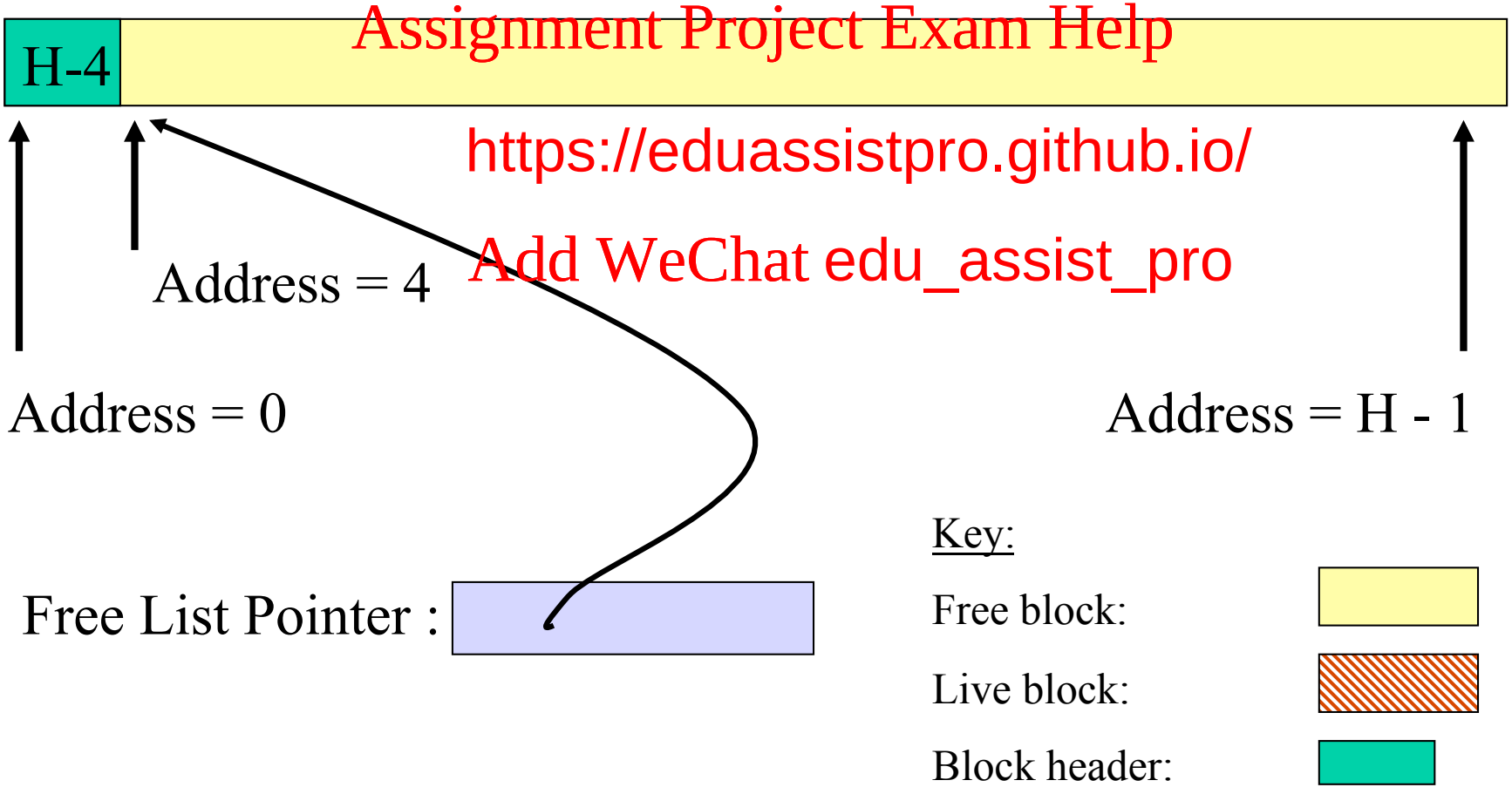
Administering a heap of H bytes

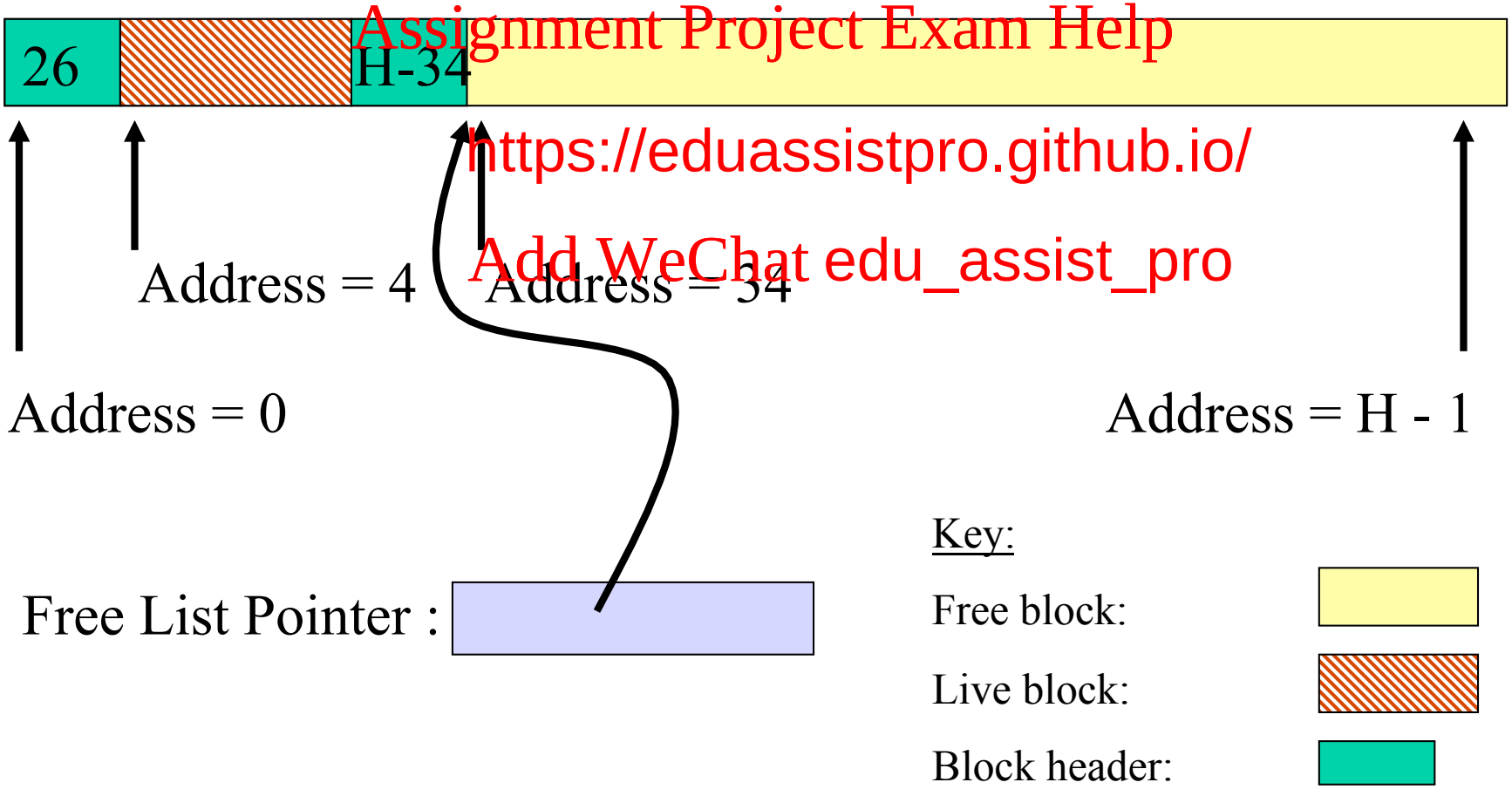


Free List Pointer :

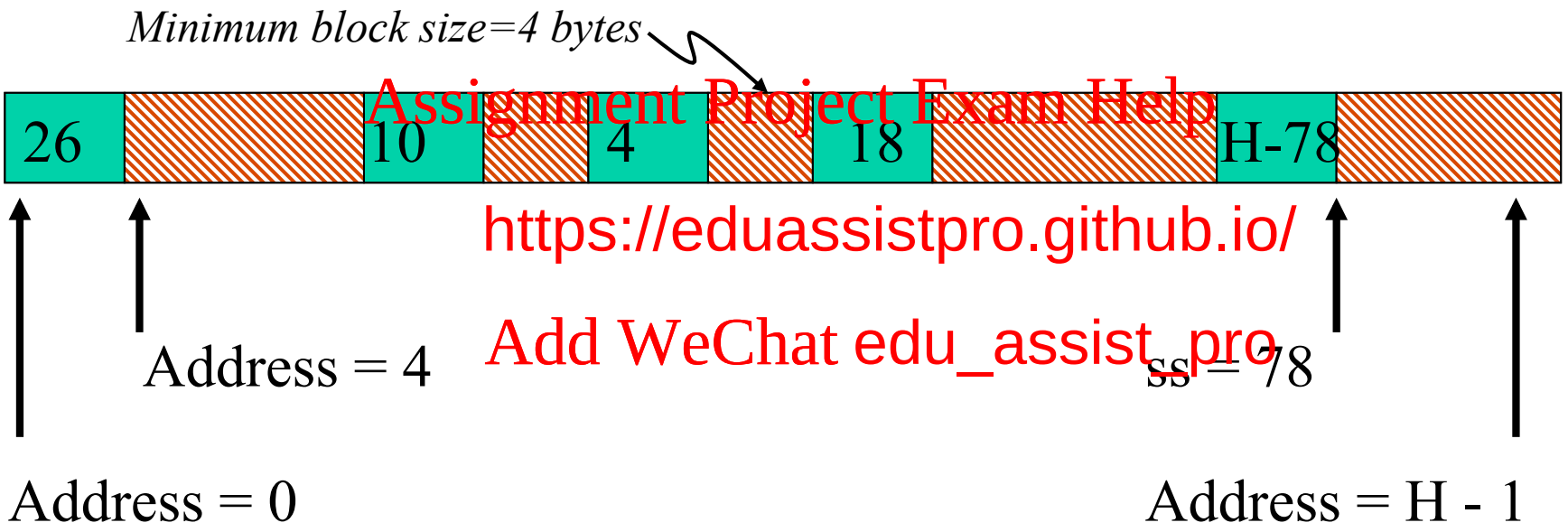
Initial Layout



After program requests block of size 26



After program requests blocks of sizes 10, 2, 18, H-78



Free List Pointer : *(null)*

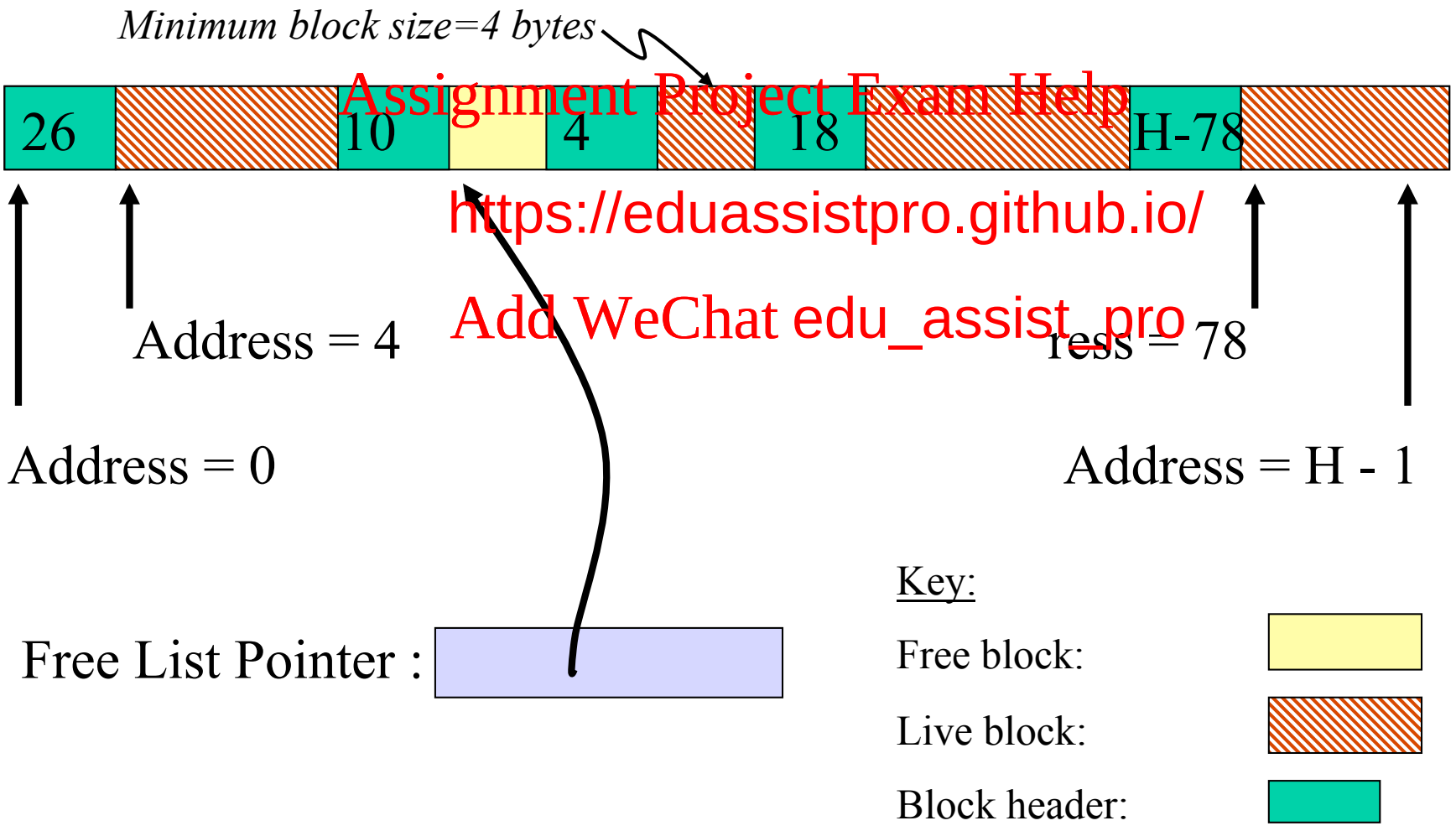
Key:

Free block:

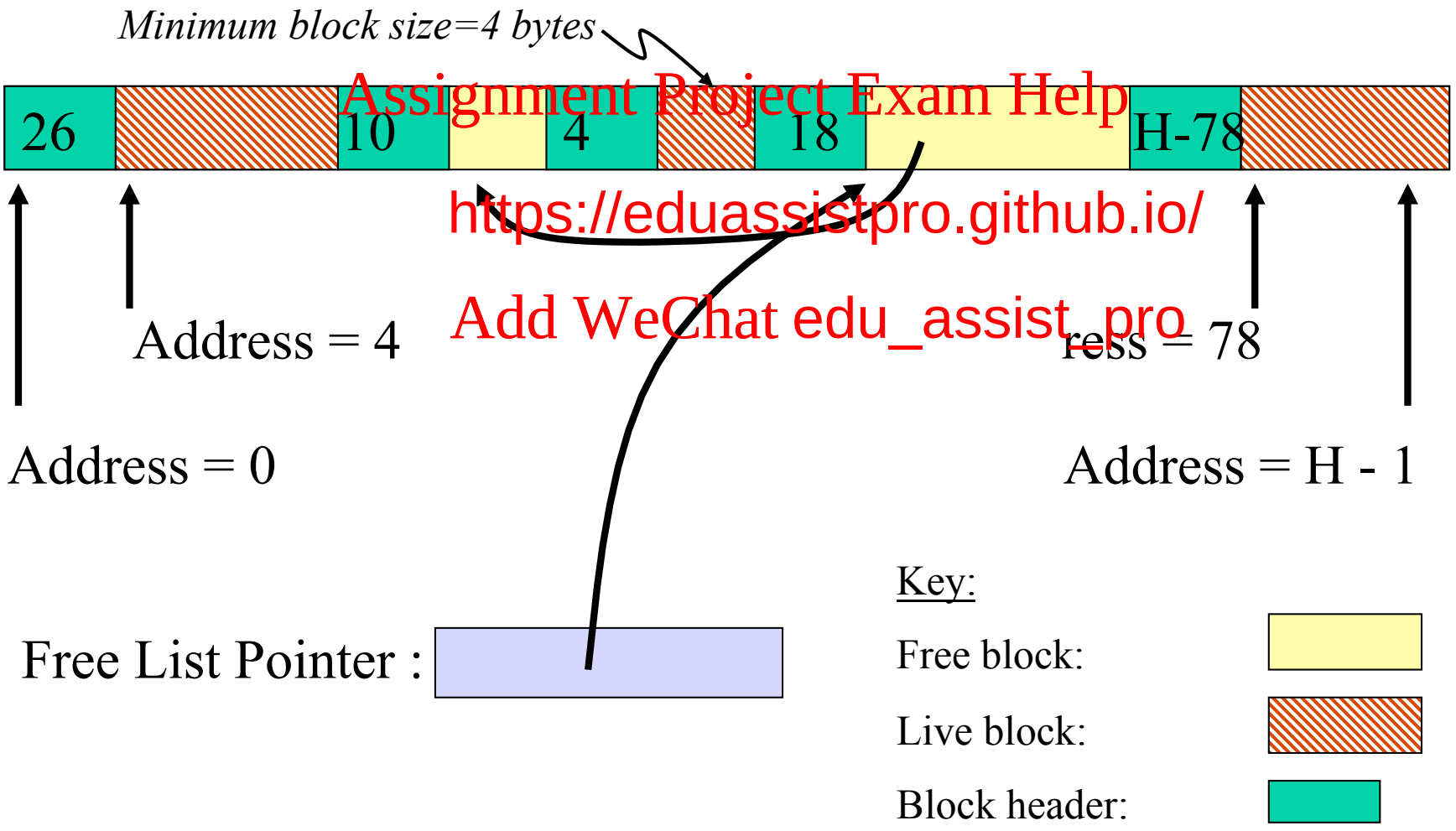
Live block:

Block header:

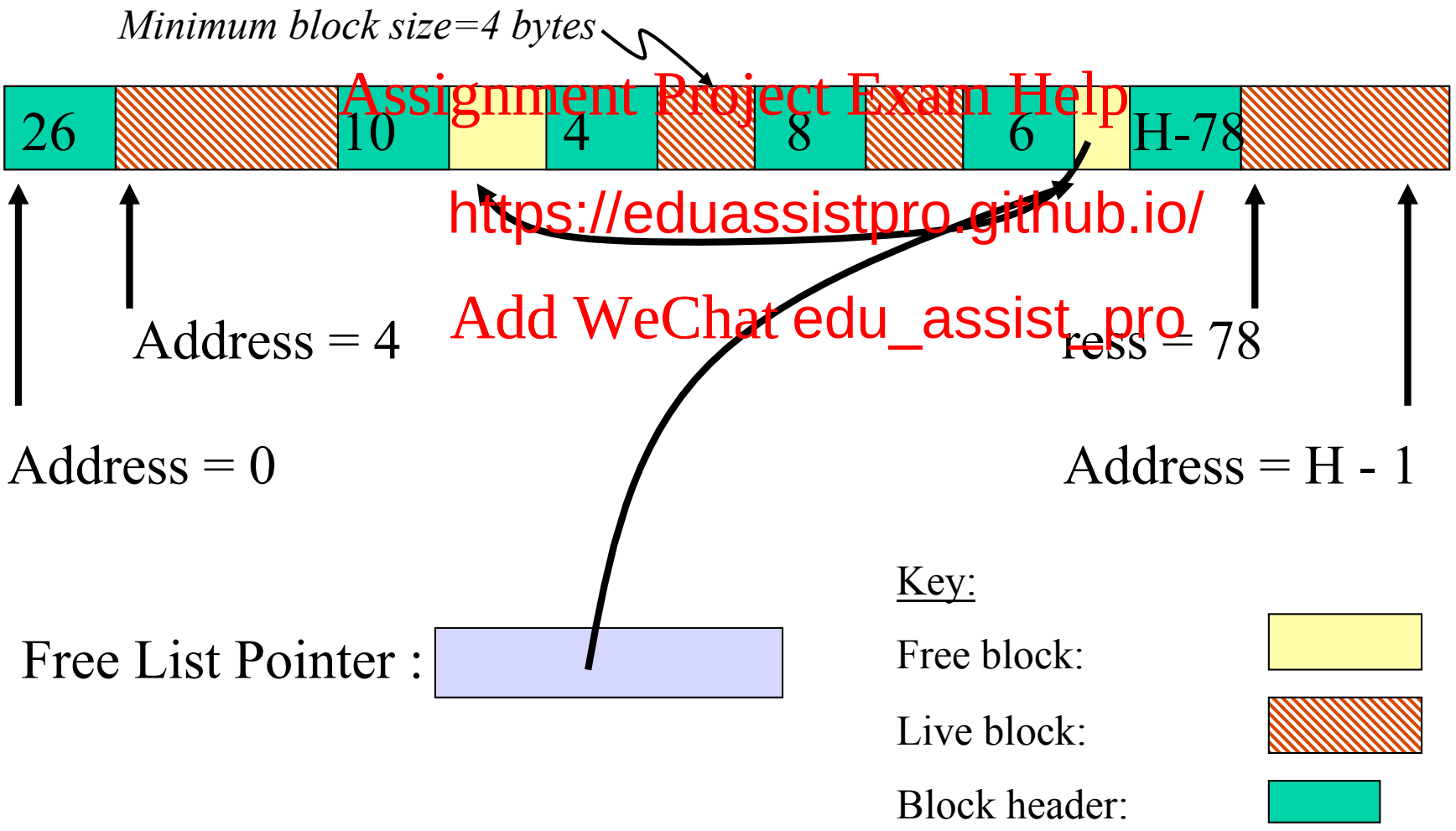
After program frees one block



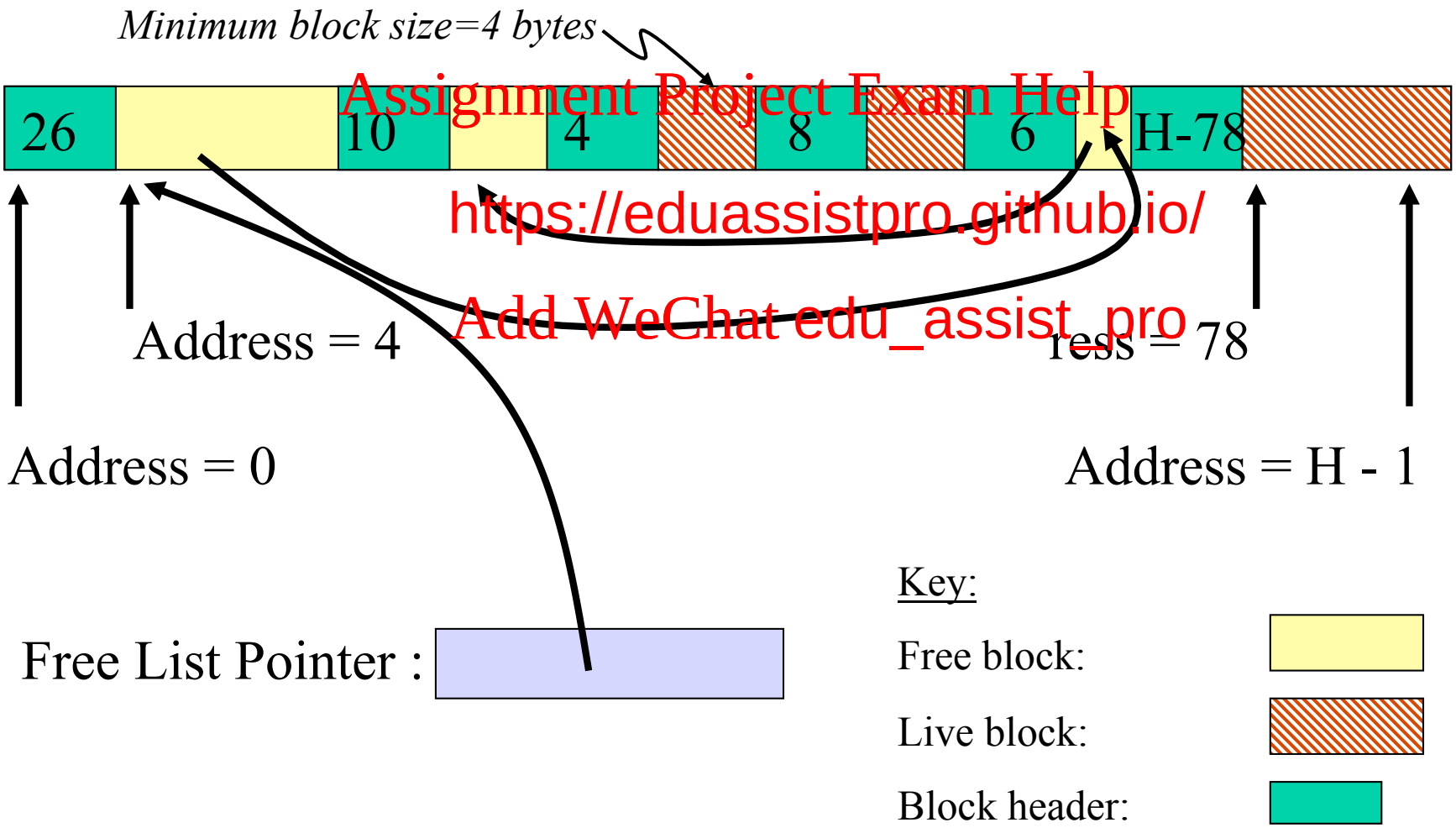
After program frees a second block (free() uses LIFO policy)



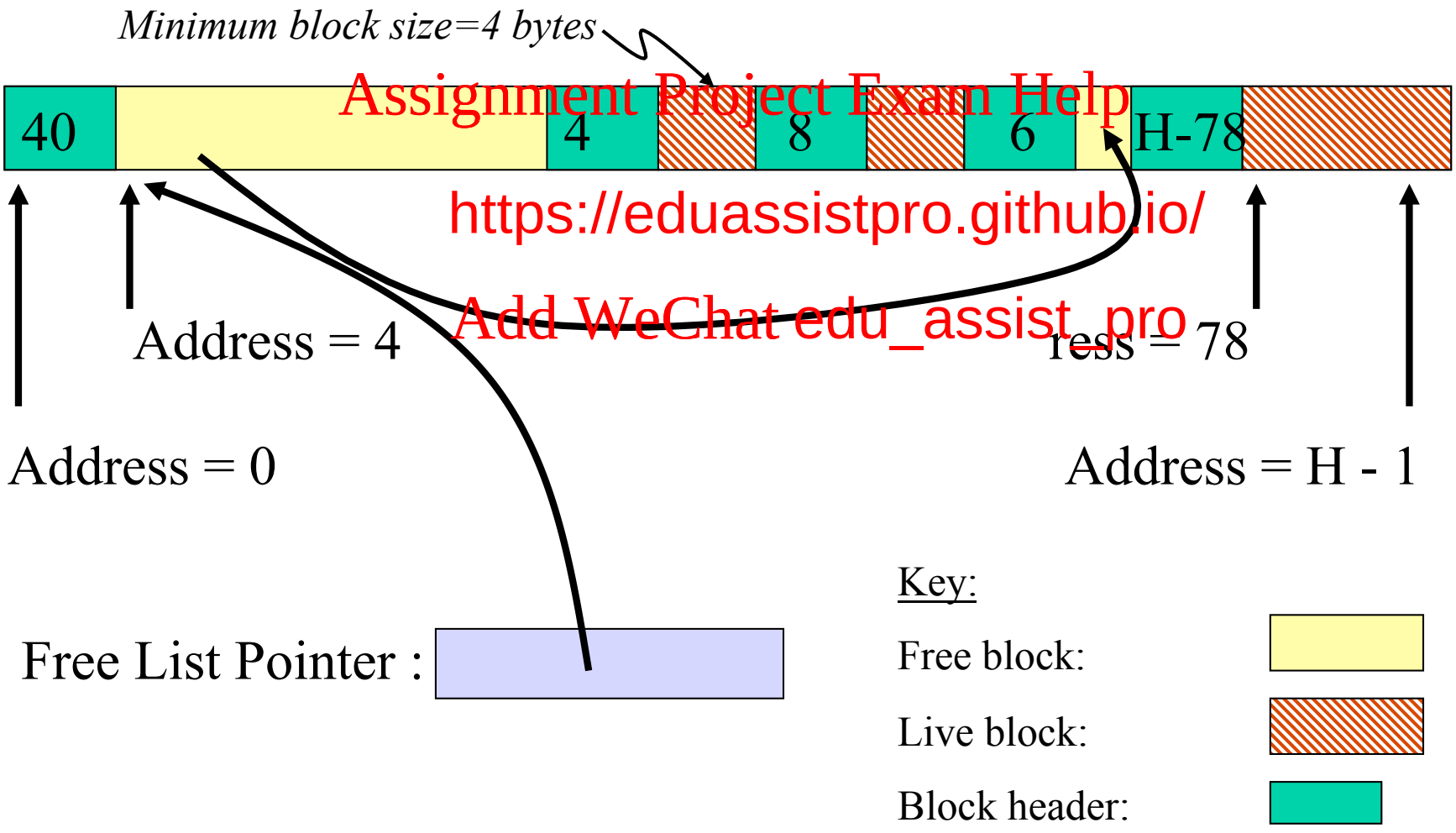
After program requests a block of size 8 (free block is split)



Program frees a block (free() uses LIFO policy – interim position)



Program frees a block (free() uses LIFO policy – final position)

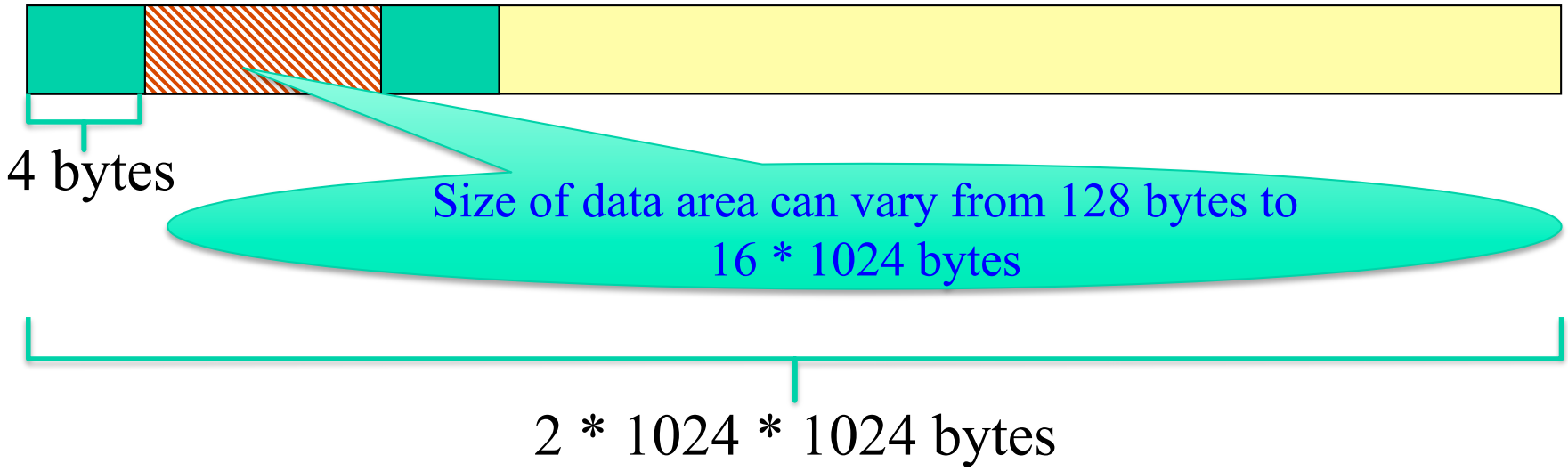


Quantitative Measurements

Quantitative Measurements

Given a heap memory with the following characteristics:

- heap size = 2Mbytes
- block header size = 4 bytes
- minimum allo
- maximum allo



- What is the maximum number of live blocks that can exist in this heap?

= heapsize ÷ minblocksize

= 2 × 1024 × 1024 ÷ (128 + 4)

= 1 whole number)

- What I the minimum number ks that can exist in this heap if there are ks?

= heapsize ÷ maxblocksize

= 2 × 1024 × 1024 ÷ (16×1024 + 4)

= 128 (NB must be a whole number)

Coalescing with Boundary Tags

- What is the minimum information that needs to be stored in each block header and what extra information would each header need to include in order to support coalescing?

The minimu to be stored
in each bloc e data area.

For coalescing, the header ntain:

(i) a FREE BIT to say whether this block is live or free

(ii) The SIZE & the FREE BIT for the PREVIOUS BLOCK (this essentially implements Knuth's boundary tags)

- How would the performance of the allocator be affected by the choice of (i) LIFO (ii) FIFO or (iii) Address Ordered (AO) management of the free list?

LIFO has a fast free() routine which operates in constant time (O(1)). FIFO may have a fast or slow free() routin pointer to the start of the l . However, if there is also a pointer to the free list, it will be fast – O(1). AO requires that blocks be inserted into the free list in address order - this will cause the free() routine to be slow – O(n). However coalescing may be faster for AO than either LIFO or FIFO if the free list is single-linked.

In the exam, say WHY

What problem is solved by Knuth's boundary tags mechanism? Give a detailed explanation of how boundary tags could be implemented in the above heap to provide a comprehensive solution to the stated problem. Specific attention should be paid to the contents of the block headers, the procedures involved, and the performance implications.

- *Knuth's boundary tags support **coalescing**.*
- *Coalescing is a process that, when a block is freed, checks the block's neighbours in memory to see whether one or both are free.*
  - *Note that free blocks that are neighbours in physical memory will not necessarily be neighbours on the free list; the exception is where AO management is used.*
  - *If AO management* *cks in memory will also be neighbouring free*
- *Any adjacent free blocks are coalesced in* *block.*
- *Information about the next block can be obtained by using the current block size to jump to the header for the next block.*

- *A 4 byte header can be used to provide size and availability information for both the current and the previous block. This is a variation of Knuth's mechanism.*
  - *In Knuth's mechanism each block has a "low boundary tag" (the header) and a "high boundary tag" (the footer). The boundary tags contain both SIZE and AVAILABILITY information for the current block, and both boundary tags must be kept identical (changes must be made to both, or neither)*
  - *In this variant, each block header contains the low boundary tag (header) for the current block and the high boundary tag (footer) for the previous block.*
  - *The highest block ry) and the lowest block has a redundant "previous" boundary tag s block that does not exist (and is therefore set perpetually to "li re is no attempt to coalesce with a non-existent block).*
- *In a 4-byte header, a 16-bit signed short int can provide availability (using the sign bit) and size (using the absolute value with 15 bits) and is appropriate for block sizes up to 2^(15)-1 fields.*
- *If a field is a byte, that means a maximum data size of 32,767 bytes (i.e. 32Kbytes - 1) which is ample for a maximum block size of 16Kbytes.*

- *When a block is freed, the availability of the previous block is read from the current block header (or from the previous block's footer, if using Knuth tags).*
  - *If the previous block is free, the size of the previous block is used to jump to the previous header block and modify it to indicate that its size is now increased by the size of the freed block.*
- *The size of the freed block is then used to jump ahead to find the availability of the next block.*
  - *If that is free, then the header of the freed block (or the header of the previous block, if that has n ed block) is modified to indicate a new size block.*
  - *The "previous" size held in the block ally coalesced free block must also be modified.*
- *Coalescing with the previous block requires no changes to the free list.*
- *Coalescing with the next block does require a change to the free list – we must find the pointer that used to point to the next block and make it point to this block.*
  - *The performance is O(n) for a singe-linked free list and O(1) for a double-linked free list. You should convince yourself WHY this is true!*

If the maximum block size were to be increased to 127Kbytes, how would this affect the block header (assume coalescing is supported)? Suggest two solutions and state the factors that would determine which solution would be preferable in a given context.

This is another question from a past exam paper, and it als nswer, but it also re lem (what happens if the maximu size were increased to a given nu efore, the answer should be carefully explained, based on the lectures, on the "dynamic storage allocation" handout found on the Moodle page, and on the following slides.

- *An increase in the max block size to 128Kbytes means that 18 bits would be needed to represent the size and a further bit for the availability.*
  - *128 Kbytes = 128 * 1024 bytes = 131,072*
  - *17 bits can represent numbers up to a maximum 2^{17}-1, which is 131,071*
  - *So 18 bits would be needed (plus 1 bit to say whether the block is live or free)*
- *This implies that each 4byte block header would have insufficient bits to represent both size an nd the previous blocks.*
- *One solution is to increase the block hea*
  - *This would have to be at least 38 bits suming the header is byte-aligned*
  - *This would be fast, but expensive in memory.*
  - *Also, since free blocks can (through coalescing) become bigger than the maximum allocatable block size, this solution is still insufficient - it needs an escape mechanism for very large free blocks.*

- *The alternative solution is to use an escape mechanism for large sizes.*
- *The standard 4 byte header can still be used, but a special pattern (eg -1) will be used to represent "TOO BIG" and the real size can be placed in an extended header area.*
- *For free blocks, it would be possible to place the real size inside the data area of the block (in which case there will be an impact on the minimum allocatable block size), but in practice this is quite slow because an extended header must always be used for live blocks and switching between the two representations takes too long.*
- *Where an escape mechanism is used, the [illegible] be made available both in (i) the current block and (ii) the next to [illegible] scanning is able to find the real size of the previous block and then jump backwards to find the previous header).*

PERFORMANCE

- *The second solution (the escape mechanism) is preferable where a lower memory overhead is required, but is slightly slower.*
- *If time performance is more important than memory overhead then the first solution would be preferred.*