Explain this implementation of malloc from the K&R book

Asked 12 years, 1 month ago Modified 1 year, 5 months ago Viewed 38k times

41

This is an excerpt from the book on C by *Kernighan and Ritchie*. It shows how to implement a version of `malloc`. Although well commented, I am having great difficulty in understanding it. Can somebody please explain it?

```
typedef long Align; /* for alignment to long boundary */
union header { /* block header */
struct {
union header *ptr; /* next block if on free list */
unsigned size; /* size of this block */
} s;
Align x; /* force alignment of blocks */
};
typedef union header Header;

static Header base; /* empty list to get started */
static Header *freep = NULL; /* start of free list */
/* malloc: general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
   Header *p, *prevp;
   Header *morecore(unsigned);
   unsigned nunits;
   nunits = (nbytes+sizeof(Header)-1)/sizeof(header) + 1;
   if ((prevp = freep) == NULL) { /* no free list yet */
      base.s.ptr = freeptr = prevptr = &base;
      base.s.size = 0;
   }
   for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
      if (p->s.size >= nunits) { /* big enough */
        if (p->s.size == nunits) /* exactly */
           prevp->s.ptr = p->s.ptr;
        else { /* allocate tail end */
           p->s.size -= nunits;
           p += p->s.size;
           p->s.size = nunits
             }
        freep = prevp;
        return (void *)(p+1);
      }
      if (p == freep) /* wrapped around free list */
         if ((p = morecore(nunits)) == NULL)
             return NULL; /* none left */
      }
}

#define NALLOC 1024 /* minimum #units to request */
/* morecore: ask system for more memory */

static Header *morecore(unsigned nu)
{
```

```
  char *cp, *sbrk(int);
  Header *up;

  if (nu < NALLOC)
    nu = NALLOC;

  cp = sbrk(nu * sizeof(Header));

  if (cp == (char *) -1) /* no space at all */
    return NULL;

  up = (Header *) cp;
  up->s.size = nu;
  free((void *)(up+1));

  return freep;
}

/* free: put block ap in free list */
void free(void *ap) {
  Header *bp, *p;
  bp = (Header *)ap - 1; /* point to block header */
  for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
    if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
      break; /* freed block at start or end of arena */
  if (bp + bp->size == p->s.ptr) {
    bp->s.size += p->s.ptr->s.size;
    bp->s.ptr = p->s.ptr->s.ptr;
  } else
      bp->s.ptr = p->s.ptr;

  if (p + p->size == bp) {
    p->s.size += bp->s.size;
    p->s.ptr = bp->s.ptr;
  } else
    p->s.ptr = bp;
  freep = p;
}
```

c pointers memory-management malloc kernighan-and-ritchie

Share

Improve this question

Follow

edited Jul 12, 2023 at 2:58

Lover of Structure

1,818 ● 3 ● 13 ● 28

asked Oct 31, 2012 at 8:45

NedStarkOfWinterfell

5,107 ● 28 ● 112 ● 203

2 I've got my K&R 2nd edition in front of me - a pretty early printing I would imagine - and it doesn't contain some of the problems the accepted answer refers to. Can I ask which edition you used and whether you typed the code in by hand? – JulianSymes Nov 27, 2013 at 10:54

1 Maybe framing specific questions (eg. why and how exactly are the blocks aligned?) will result in more helpful answers? – Emaad Ahmed Manzoor Sep 10, 2015 at 2:38

I see this in my copy of K&R 2nd ed. ISBN 0-13-110362-8. The code is presented on pg.185 in Section 8.7: Example-A Storage Allocator and is split across several paragraphs of explanation. – Lorem Ipsum Dec 11, 2021 at 20:44

1 @LoremIpsum - no, you don't. I am looking at the same edition. Some of the typos in the OP's code are not from the book - the OP added them himself (or he used a different edition). In any case, K&R say that they **tested** their code. I believe them - so I don't think there are any typos in the code (or not ones that would result in compiler errors, anyway). – user5683823 Mar 5, 2022 at 5:32

@mathguy, fair enough. There are typos in the OP's code at lines 21, 73, and 79. On line 21, `freeptr` and `prevptr` should be `freep` and `prevp`, respectively. On line 73, `bp->size` should be `bp->s.size`. Similarly, line 79 should be `p->s.size`, not `p->size`. – Lorem Ipsum Mar 7, 2022 at 19:26

4 Answers

Sorted by: Highest score (default)

64

Ok, what we have here is a chunk of really poorly written code. What I will do in this post could best be described as software archaeology.

Step 1: fix the formatting.

The indention and compact format doesn't do anyone any good. Various spaces and empty rows need to be inserted. The comments could be written in more readable ways. I'll start by fixing that.

At the same time I'm changing the brace style from K&R style - please note that the K&R brace style is acceptable, this is merely a personal preference of mine. Another personal preference is to write the * for pointers next to the type pointed at. I'll not argue about (subjective) style matters here.

Also, the type definition of `Header` is completely unreadable, it needs a drastic fix.

And I spotted something completely obscure: they seem to have declared a function prototype inside the function. `Header* morecore(unsigned);`. This is very old and very poor style, and I'm not sure if C even allows it any longer. Lets just remove that line, whatever that function does, it will have to be defined elsewhere.

```
typedef long Align;                     /* for alignment to long boundary */

typedef union header                    /* block header */
{
  struct
  {
    union header *ptr;                  /* next block if on free list */
    unsigned size;                      /* size of this block */
  } s;

  Align x;                              /* force alignment of blocks */
```

```
} Header;


static Header base;                     /* empty list to get started */
static Header* freep = NULL;            /* start of free list */


/* malloc: general-purpose storage allocator */
void* malloc (unsigned nbytes)
{
  Header*   p;
  Header*   prevp;
  unsigned  nunits;

  nunits = (nbytes + sizeof(Header) - 1) / sizeof(header) + 1;

  if ((prevp = freep) == NULL)          /* no free list yet */
  {
    base.s.ptr = freeptr = prevptr = &base;
    base.s.size = 0;
  }

  for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr)
  {
    if (p->s.size >= nunits)            /* big enough */
    {
      if (p->s.size == nunits)          /* exactly */
        prevp->s.ptr = p->s.ptr;
      else                              /* allocate tail end */
      {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits
      }

      freep = prevp;
      return (void *)(p+1);
    }

    if (p == freep)                     /* wrapped around free list */
      if ((p = morecore(nunits)) == NULL)
        return NULL;                    /* none left */
  }
}
```

Ok now we might actually be able to read the code.

Step 2: weed out widely-recognized bad practice.

This code is filled with things that are nowadays regarded as bad practice. They need to be removed, since they jeopardize the safety, readability and maintenance of the code. If you want a reference to an authority preaching the same practices as me, check out the widely-recognized coding standard MISRA-C.

I have spotted and removed the following bad practices:

1) Just typing `unsigned` in the code could lead to be confusion: was this a typo by the programmer or was the intention to write `unsigned int`? We should replace all `unsigned` with `unsigned int`. But as we do that, we find that it is used in this context to give the size of various binary data. The correct type to use for such matters is the C standard type `size_t`. This is essentially just an unsigned int as well, but it is guaranteed to be "large enough" for the particular platform. The `sizeof` operator returns a result of type `size_t` and if we look at the C standard's definition of the real malloc, it is `void *malloc(size_t size);`. So `size_t` is the most correct type to use.

2) It is a bad idea to use the same name for our own malloc function as the one residing in stdlib.h. Should we need to include stdlib.h, things will get messy. As a rule of thumb, never use identifier names of C standard library functions in your own code. I'll change the name to kr_malloc.

3) The code is abusing the fact that all static variables are guaranteed to be initialized to zero. This is well-defined by the C standard, but a rather subtle rule. Lets initialize all statics explicitly, to show that we haven't forgotten to init them by accident.

4) Assignment inside conditions is dangerous and hard to read. This should be avoided if possible, since it can also lead to bugs, such as the classic = vs == bug.

5) Multiple assignments on the same row is hard to read, and also possibly dangerous, because of the order of evaluation.

6) Multiple declarations on the same row is hard to read, and dangerous, since it could lead to bugs when mixing data and pointer declarations. Always declare each variable on a row of its own.

7) Always uses braces after every statement. Not doing so will lead to bugs bugs bugs.

8) Never type cast from a specific pointer type to void*. It is unnecessary in C, and could hide away bugs that the compiler would otherwise have detected.

9) Avoid using multiple return statements inside a function. Sometimes they lead to clearer code, but in most cases they lead to spaghetti. As the code stands, we can't change that without rewriting the loop though, so I will fix this later.

10) Keep for loops simple. They should contain one init statement, one loop condition and one iteration, nothing else. This for loop, with the comma operator and everything, is very obscure. Again, we spot a need to rewrite this loop into something sane. I'll do this next, but for now we have:

```
typedef long Align;                     /* for alignment to long boundary */

typedef union header                    /* block header */
```

```
{
  struct
  {
    union header *ptr;                  /* next block if on free list */
    size_t size;                        /* size of this block */
  } s;

  Align x;                              /* force alignment of blocks */

} Header;


static Header base = {0};               /* empty list to get started */
static Header* freep = NULL;            /* start of free list */


/* malloc: general-purpose storage allocator */
void* kr_malloc (size_t nbytes)
{
  Header*  p;
  Header*  prevp;
  size_t   nunits;

  nunits = (nbytes + sizeof(Header) - 1) / sizeof(header) + 1;

  prevp = freep;
  if (prevp == NULL)                    /* no free list yet */
  {
    base.s.ptr  = &base;
    freeptr     = &base;
    prevptr     = &base;
    base.s.size = 0;
  }

  for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr)
  {
    if (p->s.size >= nunits)            /* big enough */
    {
      if (p->s.size == nunits)          /* exactly */
      {
        prevp->s.ptr = p->s.ptr;
      }
      else                              /* allocate tail end */
      {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits
      }

      freep = prevp;
      return p+1;
    }

    if (p == freep)                     /* wrapped around free list */
    {
      p = morecore(nunits);
      if (p == NULL)
      {
        return NULL;                    /* none left */
      }
    }
```

```
  } /* for */
}
```

Step 3: rewrite the obscure loop.

For the reasons mentioned earlier. We can see that this loop goes on forever, it terminates by returning from the function, either when the allocation is done, or when there is no memory left. So lets create that as a loop condition, and lift out the return to the end of the function where it should be. And lets get rid of that ugly comma operator.

I'll introduce two new variables: one result variable to hold the resulting pointer, and another to keep track of whether the loop should continue or not. I'll blow K&R's minds by using the `bool` type, which is part of the C language since 1999.

(I hope I haven't altered the algorithm with this change, I believe I haven't)

```
#include <stdbool.h>

typedef long Align;                     /* for alignment to long boundary */

typedef union header                    /* block header */
{
  struct
  {
    union header *ptr;                  /* next block if on free list */
    size_t size;                        /* size of this block */
  } s;

  Align x;                              /* force alignment of blocks */

} Header;


static Header base = {0};               /* empty list to get started */
static Header* freep = NULL;            /* start of free list */


/* malloc: general-purpose storage allocator */
void* kr_malloc (size_t nbytes)
{
  Header*  p;
  Header*  prevp;
  size_t   nunits;
  void*    result;
  bool     is_allocating;

  nunits = (nbytes + sizeof(Header) - 1) / sizeof(header) + 1;

  prevp = freep;
  if (prevp == NULL)                    /* no free list yet */
  {
    base.s.ptr  = &base;
    freeptr     = &base;
    prevptr     = &base;
```

```
    base.s.size = 0;
  }

  is_allocating = true;
  for (p = prevp->s.ptr; is_allocating; p = p->s.ptr)
  {
    if (p->s.size >= nunits)             /* big enough */
    {
      if (p->s.size == nunits)           /* exactly */
      {
        prevp->s.ptr = p->s.ptr;
      }
      else                               /* allocate tail end */
      {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits
      }

      freep = prevp;
      result = p+1;
      is_allocating = false;             /* we are done */
    }

    if (p == freep)                      /* wrapped around free list */
    {
      p = morecore(nunits);
      if (p == NULL)
      {
        result = NULL;                   /* none left */
        is_allocating = false;
      }
    }
    prevp = p;
  } /* for */

  return result;
}
```

Step 4: make this crap compile.

Since this is from K&R, it is filled with typos. `sizeof(header)` should be `sizeof(Header)`. There are missing semi-colons. They use different names freep, prevp versus freeptr, prevptr, but clearly mean the same variable. I believe the latter were actually better names, so lets use those.

```
#include <stdbool.h>

typedef long Align;                      /* for alignment to long boundary */

typedef union header                     /* block header */
{
  struct
  {
    union header *ptr;                   /* next block if on free list */
    size_t size;                         /* size of this block */
  } s;
```

```
  Align x;                                /* force alignment of blocks */

} Header;


static Header base = {0};                 /* empty list to get started */
static Header* freeptr = NULL;            /* start of free list */


/* malloc: general-purpose storage allocator */
void* kr_malloc (size_t nbytes)
{
  Header*  p;
  Header*  prevptr;
  size_t   nunits;
  void*    result;
  bool     is_allocating;

  nunits = (nbytes + sizeof(Header) - 1) / sizeof(Header) + 1;

  prevptr = freeptr;
  if (prevptr == NULL)                    /* no free list yet */
  {
    base.s.ptr  = &base;
    freeptr     = &base;
    prevptr     = &base;
    base.s.size = 0;
  }

  is_allocating = true;
  for (p = prevptr->s.ptr; is_allocating; p = p->s.ptr)
  {
    if (p->s.size >= nunits)              /* big enough */
    {
      if (p->s.size == nunits)            /* exactly */
      {
        prevptr->s.ptr = p->s.ptr;
      }
      else                                /* allocate tail end */
      {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }

      freeptr = prevptr;
      result = p+1;
      is_allocating = false;              /* we are done */
    }

    if (p == freeptr)                     /* wrapped around free list */
    {
      p = morecore(nunits);
      if (p == NULL)
      {
        result = NULL;                    /* none left */
        is_allocating = false;
      }
    }
    prevptr = p;
  } /* for */
```

```
    return result;
}
```

And now we have somewhat readable, maintainable code, without numerous dangerous practices, that will even compile! So now we could actually start to ponder about what the code is actually doing.

The struct "Header" is, as you might have guessed, the declaration of a node in a linked list. Each such node contains a pointer to the next one. I don't quite understand the morecore function, nor the "wrap-around", I have never used this function, nor `sbrk`. But I assume that it allocates a header as specified in this struct, and also some chunk of raw data following that header. If so, that explains why there is no actual data pointer: the data is assumed to follow the header, adjacently in memory. So for each node, we get the header, and we get a chunk of raw data following the header.

The iteration itself is pretty straight-forward, they are going through a single-linked list, one node at a time.

At the end of the loop, they set the pointer to point one past the end of the "chunk", then store that in a static variable, so that the program will remember where it previously allocated memory, next time the function is called.

They are using a trick to make their header end up on an aligned memory address: they store all the overhead info in a union together with a variable large enough to correspond to the platform's alignment requirement. So if the size of "ptr" plus the size of "size" are too small to give the exact alignment, the union guarantees that at least sizeof(Align) bytes are allocated. I believe that this whole trick is obsolete today, since the C standard mandates automatic struct/union padding.

Share

Improve this answer

Follow

edited Nov 25, 2013 at 12:06

answered Oct 31, 2012 at 10:32

Lundin

212k ● 45 ● 270 ● 426

18 Most of the bad practices you mentioned aren't, they're language features. I kind of agree with #1; #2 is irrelevant, and the rest is a matter of style. – netcoder Oct 31, 2012 at 13:57

22 In my 25+ years of coding, this is the first time I've ever heard K&R called "incredibly hyped up" and flawed. – Rob Oct 31, 2012 at 13:59

30 @Rob Are you also using 25+ years old compilers? 25+ year old OS? On a 25+ year old computer? There is plenty of perfectly valid criticism against the book if you only look around. If you are going to down vote me just because I told you that the sun is the center of the solar system, and not the earth, at least provide some rationale over why you think I'm wrong. I'd just love to hear your logical reasoning over why the original code is oh-so-good. It will even

enforce you to make your own opinion about the book, instead of following the convenient bandwagon. – Lundin Oct 31, 2012 at 14:21

9 @Cupidvogel: Spreading information as factual when it is totally subjective is a good enough reason for me. – netcoder Oct 31, 2012 at 20:07

22 and we never did get to explaining how the code actually works – pm100 Oct 8, 2014 at 23:32

47

I'm studying K&R as I'd imagine OP was when he asked this question, and I came here because I also found these implementations to be confusing. While the accepted answer is very detailed and helpful, I tried to take a different tack which was to understand the code as it was originally written - I've gone through the code and added comments to the sections of the code that were difficult to me. This includes code for the other routines in the section (which are the functions `free` and `memcore` - I've renamed them `kandr_malloc` and `kandr_free` to avoid conflicts with the stdlib). I thought I would leave this here as a supplement to the accepted answer, for other students who may find it helpful.

I acknowledge that the comments in this code are excessive. Please know that I am only doing this as a learning exercise and I am not proposing that this is a good way to actually write code.

I took the liberty of changing some variable names to ones that seemed more intuitive to me; other than that the code is essentially left intact. It seems to compile and run fine for the test programs that I used, although valgrind had complaints for some applications.

Also: some of the text in the comments is lifted directly from K&R or the man pages - I do not intend to take any credit for these sections.

```
#include <unistd.h>  // sbrk

#define NALLOC 1024  // Number of block sizes to allocate on call to sbrk
#ifdef NULL
#undef NULL
#endif
#define NULL 0


// long is chosen as an instance of the most restrictive alignment type
typedef long Align;

/* Construct Header data structure.  To ensure that the storage returned by
 * kandr_malloc is aligned properly for the objects that are stored in it, all
 * blocks are multiples of the header size, and the header itself is aligned
 * properly.  This is achieved through the use of a union; this data type is
big
 * enough to hold the "widest" member, and the alignment is appropriate for all
 * of the types in the union.  Thus by including a member of type Align, which
```

```
 * is an instance of the most restrictive type, we guarantee that the size of
 * Header is aligned to the worst-case boundary.  The Align field is never
used;
 * it just forces each header to the desired alignment.
 */
union header {
  struct {
    union header *next;
    unsigned size;
  } s;

  Align x;
};
typedef union header Header;


static Header base;           // Used to get an initial member for free list
static Header *freep = NULL;  // Free list starting point


static Header *morecore(unsigned nblocks);
void kandr_free(void *ptr);




void *kandr_malloc(unsigned nbytes) {

  Header *currp;
  Header *prevp;
  unsigned nunits;

  /* Calculate the number of memory units needed to provide at least nbytes of
   * memory.
   *
   * Suppose that we need n >= 0 bytes and that the memory unit sizes are b > 0
   * bytes.  Then n / b (using integer division) yields one less than the
number
   * of units needed to provide n bytes of memory, except in the case that n is
   * a multiple of b; then it provides exactly the number of units needed.  It
   * can be verified that (n - 1) / b provides one less than the number of
units
   * needed to provide n bytes of memory for all values of n > 0.  Thus ((n -
1)
   * / b) + 1 provides exactly the number of units needed for n > 0.
   *
   * The extra sizeof(Header) in the numerator is to include the unit of memory
   * needed for the header itself.
   */
  nunits = ((nbytes + sizeof(Header) - 1) / sizeof(Header)) + 1;

  // case: no free list yet exists; we have to initialize.
  if (freep == NULL) {

    // Create degenerate free list; base points to itself and has size 0
    base.s.next = &base;
    base.s.size = 0;

    // Set free list starting point to base address
    freep = &base;
  }
```

```
  /* Initialize pointers to two consecutive blocks in the free list, which we
   * call prevp (the previous block) and currp (the current block)
   */
  prevp = freep;
  currp = prevp->s.next;

  /* Step through the free list looking for a block of memory large enough to
   * fit nunits units of memory into.  If the whole list is traversed without
   * finding such a block, then morecore is called to request more memory from
   * the OS.
   */
  for (; ; prevp = currp, currp = currp->s.next) {

    /* case: found a block of memory in free list large enough to fit nunits
     * units of memory into.  Partition block if necessary, remove it from the
     * free list, and return the address of the block (after moving past the
     * header).
     */
    if (currp->s.size >= nunits) {

      /* case: block is exactly the right size; remove the block from the free
       * list by pointing the previous block to the next block.
       */
      if (currp->s.size == nunits) {
    /* Note that this line wouldn't work as intended if we were down to only
     * 1 block.  However, we would never make it here in that scenario
     * because the block at &base has size 0 and thus the conditional will
     * fail (note that nunits is always >= 1).  It is true that if the block
     * at &base had combined with another block, then previous statement
     * wouldn't apply - but presumably since base is a global variable and
     * future blocks are allocated on the heap, we can be sure that they
     * won't border each other.
     */
    prevp->s.next = currp->s.next;
      }
      /* case: block is larger than the amount of memory asked for; allocate
       * tail end of the block to the user.
       */
      else {
    // Changes the memory stored at currp to reflect the reduced block size
    currp->s.size -= nunits;
    // Find location at which to create the block header for the new block
    currp += currp->s.size;
    // Store the block size in the new header
    currp->s.size = nunits;
      }

      /* Set global starting position to the previous pointer.  Next call to
       * malloc will start either at the remaining part of the partitioned
block
       * if a partition occurred, or at the block after the selected block if
       * not.
       */
      freep = prevp;

      /* Return the location of the start of the memory, i.e. after adding one
       * so as to move past the header
       */
      return (void *) (currp + 1);

    } // end found a block of memory in free list case
```

```
    /* case: we've wrapped around the free list without finding a block large
     * enough to fit nunits units of memory into.  Call morecore to request
that
     * at least nunits units of memory are allocated.
     */
    if (currp == freep) {
      /* morecore returns freep; the reason that we have to assign currp to it
       * again (since we just tested that they are equal), is that there is a
       * call to free inside of morecore that can potentially change the value
       * of freep.  Thus we reassign it so that we can be assured that the
newly
       * added block is found before (currp == freep) again.
       */
      if ((currp = morecore(nunits)) == NULL) {
    return NULL;
      }
    } // end wrapped around free list case
  } // end step through free list looking for memory loop
}




static Header *morecore(unsigned nunits) {

  void *freemem;    // The address of the newly created memory
  Header *insertp;  // Header ptr for integer arithmatic and constructing
header

  /* Obtaining memory from OS is a comparatively expensive operation, so obtain
   * at least NALLOC blocks of memory and partition as needed
   */
  if (nunits < NALLOC) {
    nunits = NALLOC;
  }

  /* Request that the OS increment the program's data space.  sbrk changes the
   * location of the program break, which defines the end of the process's data
   * segment (i.e., the program break is the first location after the end of
the
   * uninitialized data segment).  Increasing the program break has the effect
   * of allocating memory to the process.  On success, brk returns the previous
   * break - so if the break was increased, then this value is a pointer to the
   * start of the newly allocated memory.
   */
  freemem = sbrk(nunits * sizeof(Header));
  // case: unable to allocate more memory; sbrk returns (void *) -1 on error
  if (freemem == (void *) -1) {
    return NULL;
  }

  // Construct new block
  insertp = (Header *) freemem;
  insertp->s.size = nunits;

  /* Insert block into the free list so that it is available for malloc.  Note
   * that we add 1 to the address, effectively moving to the first position
   * after the header data, since of course we want the block header to be
   * transparent for the user's interactions with malloc and free.
   */
  kandr_free((void *) (insertp + 1));
```

```
  /* Returns the start of the free list; recall that freep has been set to the
   * block immediately preceeding the newly allocated memory (by free).  Thus
by
   * returning this value the calling function can immediately find the new
   * memory by following the pointer to the next block.
   */
  return freep;
}




void kandr_free(void *ptr) {

  Header *insertp, *currp;

  // Find address of block header for the data to be inserted
  insertp = ((Header *) ptr) - 1;

  /* Step through the free list looking for the position in the list to place
   * the insertion block.  In the typical circumstances this would be the block
   * immediately to the left of the insertion block; this is checked for by
   * finding a block that is to the left of the insertion block and such that
   * the following block in the list is to the right of the insertion block.
   * However this check doesn't check for one such case, and misses another.
We
   * still have to check for the cases where either the insertion block is
   * either to the left of every other block owned by malloc (the case that is
   * missed), or to the right of every block owned by malloc (the case not
   * checked for).  These last two cases are what is checked for by the
   * condition inside of the body of the loop.
   */
  for (currp = freep; !((currp < insertp) && (insertp < currp->s.next)); currp
= currp->s.next) {

    /* currp >= currp->s.ptr implies that the current block is the rightmost
     * block in the free list.  Then if the insertion block is to the right of
     * that block, then it is the new rightmost block; conversely if it is to
     * the left of the block that currp points to (which is the current
leftmost
     * block), then the insertion block is the new leftmost block.  Note that
     * this conditional handles the case where we only have 1 block in the free
     * list (this case is the reason that we need >= in the first test rather
     * than just >).
     */
    if ((currp >= currp->s.next) && ((currp < insertp) || (insertp < currp-
>s.next))) {
      break;
    }
  }

  /* Having found the correct location in the free list to place the insertion
   * block, now we have to (i) link it to the next block, and (ii) link the
   * previous block to it.  These are the tasks of the next two if/else pairs.
   */

  /* case: the end of the insertion block is adjacent to the beginning of
   * another block of data owned by malloc.  Absorb the block on the right into
   * the block on the left (i.e. the previously existing block is absorbed into
   * the insertion block).
   */
  if ((insertp + insertp->s.size) == currp->s.next) {
```

```
    insertp->s.size += currp->s.next->s.size;
    insertp->s.next = currp->s.next->s.next;
  }
  /* case: the insertion block is not left-adjacent to the beginning of another
   * block of data owned by malloc.  Set the insertion block member to point to
   * the next block in the list.
   */
  else {
    insertp->s.next = currp->s.next;
  }

  /* case: the end of another block of data owned by malloc is adjacent to the
   * beginning of the insertion block.  Absorb the block on the right into the
   * block on the left (i.e. the insertion block is absorbed into the
preceeding
   * block).
   */
  if ((currp + currp->s.size) == insertp) {
    currp->s.size += insertp->s.size;
    currp->s.next = insertp->s.next;
  }
  /* case: the insertion block is not right-adjacent to the end of another
block
   * of data owned by malloc.  Set the previous block in the list to point to
   * the insertion block.
   */
  else {
    currp->s.next = insertp;
  }

  /* Set the free pointer list to start the block previous to the insertion
   * block.  This makes sense because calls to malloc start their search for
   * memory at the next block after freep, and the insertion block has as good
a
   * chance as any of containing a reasonable amount of memory since we've just
   * added some to it.  It also coincides with calls to morecore from
   * kandr_malloc because the next search in the iteration looks at exactly the
   * right memory block.
   */
  freep = currp;
}
```

Share Improve this answer Follow

answered Apr 9, 2016 at 2:40

dpritch

1,269 ● 14 ● 17

4 Good lord, that is such a thorough and detailed answer! Thanks! I am broke now, but one day I will be rich (with SO credits), and then I will award you a thoroughly deserved bounty.. :) Having said that, although it is well commented, I am still having problems with the utility of the `Align` word and what it does, and what you mean by alignment. Can you explain a bit more? – NedStarkOfWinterfell Apr 9, 2016 at 11:22

1 I'm only learning these concepts myself right now, so I can only say what I think is happening. Computer architectures operate on words, i.e. 32- or 64- bit segments of data. The `malloc` routine here operates on multiples of a particular unit size of memory, defined as `sizeof(Header)`. When we allocate data we need it to start and end at word boundaries. So I think they choose a data type that has a full word length, which guarantees that

`sizeof(Header)` is a multiple of the word size, and consequently that `malloc` allocates data that starts and ends on the word boundaries. – dpritch Apr 9, 2016 at 19:20

@AttitudeMonger The authors also explain Align and refer to as the machine's "most restrictive type", a term which is also discussed in this SO Question. – Brandin Aug 17, 2016 at 13:57

1 Thanks @dpritch for this awesome clarification! I was having a tough time to understand the assignment of `nunits` to the `currp->s.size`. It's nice when such answers come out here :) – Niloct Jun 5, 2017 at 1:26

1 @bag: Yes, after `currp += currp->s.size`, the new `currp->s.size` will now have an unknown value (basically whatever was last written to that memory address; *not* necessarily `NULL`). But that doesn't matter, since we assign a known value (`nunits`) there on the very next line, overwriting whatever garbage was there before. – Ilmari Karonen Mar 15, 2018 at 14:26

16

The basic of malloc()

In Linux, there are two typical ways to request memory: sbrk and mmap. These system calls have severe limitations on frequent small allocations. malloc() is a library function to address this issue. It requests large chunks of memory with sbrk/mmap and returns small memory blocks inside large chunks. This is much more efficient and flexible than directly calling sbrk/mmap.

K&R malloc()

In the K&R implementation, a *core* (more commonly called *arena*) is a large chunk of memory. `morecore()` requests a core from system via `sbrk()`. When you call malloc()/free() multiple times, some blocks in the cores are used/allocated while others are free. K&R malloc stores the addresses of free blocks in a **circular** single linked list. In this list, each node is a block of free memory. The first `sizeof(Header)` bytes keep the size of the block and the pointer to the next free block. The rest of bytes in the free block are uninitialized. Different from typical lists in textbooks, nodes in the free list are just pointers to some unused areas in cores; you don't actually allocate each node except for cores. This list is the key to the understanding of the algorithm.

The following diagram shows an example memory layout with two cores/arenas. In the diagram, each character takes `sizeof(Header)` bytes. `@` is a `Header`, `+` marks allocated memory and `-` marks free memory inside cores. In the example, there are three allocated blocks and three free blocks. The three free blocks are stored in the circular list. For the three allocated blocks, only their sizes are stored in `Header`.

```
            This is core 1                                  This is core 2
```

```
@---------@+++++++++@++++++++++++        @----------@++++++++++++++++@--------
----
|                                        |                                |
p->ptr->ptr                              p = p->ptr->ptr->ptr             p->ptr
```

In your code, `freep` is an entry point to the free list. If you repeatedly follow `freep->ptr`, you will come back to `freep` – it is circular. Once you understand the circular single-linked list, the rest is relatively easy. `malloc()` finds a free block and possibly splits it. `free()` adds a free block back to the list and may merge it to adjacent free blocks. They both try to maintain the structure of the list.

Other comments on the implementation

- The code comments mentioned "wrapped around" in `malloc()`. That line happens when you have traversed the entire free list but can't find a free block larger than the requested length. In this case, you have to add a new core with `morecore()`.
- `base` is a zero-sized block that is always included in the free list. It is a trick to avoid special casing. It is not strictly necessary.
- `free()` may look a little complex because it has to consider four different cases to merge a newly freed block to other free blocks in the list. This detail is not that important unless you want to reimplement by yourself.
- This blog post explains K&R malloc in more details.

PS: K&R malloc is one of the most elegant pieces of code in my view. It was really eye opening when I first understood the code. It makes me sad that some modern programmers, not even understanding the basic of this implementation, are calling the masterpiece crap solely based on its coding style.

Share Improve this answer Follow

answered Sep 19, 2017 at 4:17

user172818

4,724 ● 1 ● 20 ● 23

4

I also found this exercise great and interesting.

In my opinion visualizing the structure may help a lot with understanding the logic - or at least this worked for me. Below is my code, which prints as much as possible about the flow of the K&R malloc.

The most significant change I made in the K&R malloc is the change of 'free' to make sure some old pointer will not be used again. Other than that I added comments and fixed some small typos.

Experimenting with NALLOC, MAXMEM and the test variables in 'main' could be also of help.

On my computer (Ubuntu 16.04.3) this compiled without errors with:

```
gcc -g -std=c99 -Wall -Wextra -pedantic-errors krmalloc.c
```

krmalloc.c :

```
#include <stdio.h>
#include <unistd.h>

typedef long Align;             /* for alignment to long boundary */
union header {                  /* block header */
    struct {
        union header *ptr;      /* next block if on free list */
        size_t size;            /* size of this block */
                                /*      including the Header itself */
                                /*      measured in count of Header chunks */
                                /*      not less than NALLOC Header's */
    } s;
    Align x;                    /* force alignment of blocks */
};
typedef union header Header;

static Header *morecore(size_t);
void *mmalloc(size_t);
void _mfree(void **);
void visualize(const char*);
size_t getfreem(void);
size_t totmem = 0;              /* total memory in chunks */

static Header base;             /* empty list to get started */
static Header *freep = NULL;    /* start of free list */

#define NALLOC 1                /* minimum chunks to request */
#define MAXMEM 2048             /* max memory available (in bytes) */

#define mfree(p) _mfree((void **)&p)

void *sbrk(__intptr_t incr);


int main(void)
{
    char *pc, *pcc, *pccc, *ps;
    long *pd, *pdd;
    int dlen = 100;
    int ddlen = 50;

    visualize("start");


    /* trying to fragment as much as possible to get a more interesting view */

    /* claim a char */
    if ((pc = (char *) mmalloc(sizeof(char))) == NULL)
```

```
        return -1;

    /* claim a string */
    if ((ps = (char *) mmalloc(dlen * sizeof(char))) == NULL)
        return -1;

    /* claim some long's */
    if ((pd = (long *) mmalloc(ddlen * sizeof(long))) == NULL)
        return -1;

    /* claim some more long's */
    if ((pdd = (long *) mmalloc(ddlen * 2 * sizeof(long))) == NULL)
        return -1;

    /* claim one more char */
    if ((pcc = (char *) mmalloc(sizeof(char))) == NULL)
        return -1;

    /* claim the last char */
    if ((pccc = (char *) mmalloc(sizeof(char))) == NULL)
        return -1;


    /* free and visualize */
    printf("\n");
    mfree(pccc);
    /*      bugged on purpose to test free(NULL) */
    mfree(pccc);
    visualize("free(the last char)");

    mfree(pdd);
    visualize("free(lot of long's)");

    mfree(ps);
    visualize("free(string)");

    mfree(pd);
    visualize("free(less long's)");

    mfree(pc);
    visualize("free(first char)");

    mfree(pcc);
    visualize("free(second char)");


    /* check memory condition */
    size_t freemem = getfreem();
    printf("\n");
    printf("--- Memory claimed  : %ld chunks (%ld bytes)\n",
                totmem, totmem * sizeof(Header));
    printf("    Free memory now : %ld chunks (%ld bytes)\n",
                freemem, freemem * sizeof(Header));
    if (freemem == totmem)
        printf("    No memory leaks detected.\n");
    else
        printf("    (!) Leaking memory: %ld chunks (%ld bytes).\n",
                    (totmem - freemem), (totmem - freemem) * sizeof(Header));

    printf("// Done.\n\n");
    return 0;
}
```

```c
/* visualize: print the free list (educational purpose) */
void visualize(const char* msg)
{
    Header *tmp;

    printf("--- Free list after \"%s\":\n", msg);

    if (freep == NULL) {                    /* does not exist */
        printf("\tList does not exist\n\n");
        return;
    }

    if  (freep == freep->s.ptr) {           /* self-pointing list = empty */
        printf("\tList is empty\n\n");
        return;
    }

    printf("  ptr: %10p size: %-3lu -->  ", (void *) freep, freep->s.size);

    tmp = freep;                            /* find the start of the list */
    while (tmp->s.ptr > freep) {            /* traverse the list */
        tmp = tmp->s.ptr;
        printf("ptr: %10p size: %-3lu -->  ", (void *) tmp, tmp->s.size);
    }
    printf("end\n\n");
}


/* calculate the total amount of available free memory */
size_t getfreem(void)
{
    if (freep == NULL)
        return 0;

    Header *tmp;
    tmp = freep;
    size_t res = tmp->s.size;

    while (tmp->s.ptr > tmp) {
        tmp = tmp->s.ptr;
        res += tmp->s.size;
    }

    return res;
}


/* mmalloc: general-purpose storage allocator */
void *mmalloc(size_t nbytes)
{
    Header *p, *prevp;
    size_t nunits;

    /* smallest count of Header-sized memory chunks */
    /*  (+1 additional chunk for the Header itself) needed to hold nbytes */
    nunits = (nbytes + sizeof(Header) - 1) / sizeof(Header) + 1;

    /* too much memory requested? */
    if (((nunits + totmem + getfreem())*sizeof(Header)) > MAXMEM) {
        printf("Memory limit overflow!\n");
```

```
        return NULL;
    }

    if ((prevp = freep) == NULL) {          /* no free list yet */
        /* set the list to point to itself */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }

    /* traverse the circular list */
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {

        if (p->s.size >= nunits) {          /* big enough */
            if (p->s.size == nunits)        /* exactly */
                prevp->s.ptr = p->s.ptr;
            else {                          /* allocate tail end */
                /* adjust the size */
                p->s.size -= nunits;
                /* find the address to return */
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }

        /* back where we started and nothing found - we need to allocate */
        if (p == freep)                     /* wrapped around free list */
            if ((p = morecore(nunits)) == NULL)
                return NULL;                /* none left */
    }
}


/* morecore: ask system for more memory */
/*      nu: count of Header-chunks needed */
static Header *morecore(size_t nu)
{
    char *cp;
    Header *up;

    /* get at least NALLOC Header-chunks from the OS */
    if (nu < NALLOC)
        nu = NALLOC;

    cp = (char *) sbrk(nu * sizeof(Header));

    if (cp == (char *) -1)                  /* no space at all */
        return NULL;

    printf("... (sbrk) claimed %ld chunks.\n", nu);
    totmem += nu;                           /* keep track of allocated memory
*/

    up = (Header *) cp;
    up->s.size = nu;

    /* add the free space to the circular list */
    void *n = (void *)(up+1);
    mfree(n);

    return freep;
```

```
}


/* mfree: put block ap in free list */
void _mfree(void **ap)
{
    if (*ap == NULL)
        return;

    Header *bp, *p;
    bp = (Header *)*ap - 1;                 /* point to block header */

    if (bp->s.size == 0 || bp->s.size > totmem) {
        printf("_mfree: impossible value for size\n");
        return;
    }

    /* the free space is only marked as free, but 'ap' still points to it */
    /* to avoid reusing this address and corrupt our structure set it to '\0'
*/
    *ap = NULL;

    /* look where to insert the free space */

    /* (bp > p && bp < p->s.ptr)    => between two nodes */
    /* (p > p->s.ptr)               => this is the end of the list */
    /* (p == p->p.str)              => list is one element only */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            /* freed block at start or end of arena */
            break;

    if (bp + bp->s.size == p->s.ptr) {      /* join to upper nbr */
    /* the new block fits perfect up to the upper neighbor */

        /* merging up: adjust the size */
        bp->s.size += p->s.ptr->s.size;
        /* merging up: point to the second next */
        bp->s.ptr = p->s.ptr->s.ptr;

    } else
        /* set the upper pointer */
        bp->s.ptr = p->s.ptr;

    if (p + p->s.size == bp) {              /* join to lower nbr */
    /* the new block fits perfect on top of the lower neighbor */

        /* merging below: adjust the size */
        p->s.size += bp->s.size;
        /* merging below: point to the next */
        p->s.ptr = bp->s.ptr;

    } else
        /* set the lower pointer */
        p->s.ptr = bp;

    /* reset the start of the free list */
    freep = p;
}
```

Share

Improve this answer

Follow

edited Jan 1, 2018 at 8:49

answered Jan 1, 2018 at 8:05

Vladimir

1,125 1 13 25