CS 240, hw5 Memory Allocation, counts for 20 points (regular hw is 10 pts.)

In C we use malloc() and free() to allocate and free blocks of memory while our program is running. This assignment shows how the malloc and free library functions could be implemented. In fact, the actual implementations are more complicated than what we have set up here. But like the functions of this assignment, they are just C-coded functions, and their data structures can be damaged by the calling program if it writes beyond the block given out, leading to quite mysterious bugs.

In this assignment, you are to write part of a dynamic storage allocation package. The package provides three function calls: void initalloc(), to initialize the data structures involved; char * alloc(int n), which returns a pointer to a block of n chars when called; and void freef(char * p), which frees the block of n chars earlier given to the data structure so that it can be given out to another. A package somewhat like this is covered in Section 8.7 of Kernighan and Ritchie -- however, be very clear that there are important differences between the two packages. The most important differences are that we do NOT try to allocate new space if we run out, we keep ALL our space to allocate in a single array, and we do not keep our free blocks in order, so we find another way to coalesce blocks when they are being freed (one which is faster than a linear search of the free list).

The algorithm you will be working on takes storage blocks from an array of ALLOCSIZE characters, and returns them to requesting callers. The main feature of this package is that these blocks can then be freed in any order and the small blocks freed will be merged back into longer blocks in the array structure. In order to perform this merge efficiently, a rather complex structure must be placed on the individual blocks passed out to callers. In particular, this means that if a caller wants to alloc(n), we must look for a block of n+k bytes, where the k bytes will contain the overhead.

In separate files in this directory are alloctest.c, alloc.c, and alloc.h. The file alloctest.c is a main() program to drive and test your alloc() and freef() functions. All of the alloc(), initalloc(), and some needed helper functions enchain() and unchain() to place blocks on the free chain of blocks are already provided. Your job is to write the freef() function. Note that each block which is handed out has information at the left (struct blockl) and at the right (struct blockr), to aid the freef() function in coalescing freed blocks. In particular, both ends have an 8-bit pattern to let us know if the block is free or used. Then if it is free, the length is immediately available, in particular so one can get back to the left end of the block from the right end. At the left end of the block is the pointer to the next and previous blocks on the freechain (in freef(), we will have to remove adjacent blocks from the chain to coalesce with the block being freed).

alloctest.c provides an interactive test facility, for example the commands:

```
a 200
a 100
```

would call alloc(100) and then alloc(200) as the 0th and 1st alloc's, and put the returned pointers in an array "holdp", in holdp[0] and holdp[1]. Then

```
f 0
```

would free the 200-byte block of the 0th alloc (pointer in holdp[0]).

The "d" command dumps the free list, following the nextp pointers starting from freep. Thus it does not warn you about problems in the prevp pointers--you could make it follow these as well.

Testalloc can also be driven by file input. A file "test0" is provided as an example and for your final run. Use UNIX redirection to make testalloc read test0.

Leave alloc.c, the executable testalloc, and a typescript showing a run of "testalloc test0" in your hw5 directory for grading. Other files that should be there are alloc.h, testalloc.c, and test0, but these should not be editted from the provided copies.

 # makefile for building cs240 alloc project programs using the

```
# tell make to use gcc as C compiler, -g -W... for C compiler flags--
# check "man gcc" to find out what -W flags do
CC = gcc
OBJS = alloctest.o alloc.o
CFLAGS = -g -Wall -Wstrict-prototypes -Wmissing-prototypes \
           -Wno-uninitialized -Wshadow -ansi

alloctest:  $(OBJS)
      $(CC) -g -o alloctest $(OBJS)

# make knows to use $(CC) and $(CFLAGS)
alloctest.o: alloctest.c alloc.h
alloc.o: alloc.c alloc.h

clean:
      rm -f *.o testalloc core
```

/* alloc.h: external specs for alloc/freef package */

```
/* call this once before any alloc()'s */
void initalloc(void);

/* allocate a block of n bytes of memory and return its address, guaranteed
 * to be a multiple of 4 ("4-byte-aligned") to make sure that the
 * provided address can be used as an int * pointer for 32-bit ints.
 */
char *alloc(int n);

/* free a previously alloc'd block */
void freef(char *p);

/* for debugging */
void dumpfree(void);
```

```
/* Package for free storage allocation:
   void initalloc()    initialize package
   char *alloc(int n)  allocate n bytes, longword aligned
   freef(char *p)      free previously alloc'd bloc

   Patrick O'Neil with help from Betty O'Neil, edited 04/15/00            */
/* external spec's--just function prototypes for initalloc,
   alloc and freef--                                                      */
#include "alloc.h"
#include <stdio.h>

/* internal def's--hidden from user of package--                          */
#define NULL 0                  /* pointer value for error               */
#define ALLOCSIZE 1000          /* size of allocation array              */
/* tag+size--assumed identical in blockr and blockl-- */
#define TAGSIZE sizeof(struct blockr)
/* minimum free block size--a blockl, then a blockr--                      */
#define MINFREESIZE (sizeof(struct blockl)+sizeof(struct blockr))
#define FREETAG 0x55            /* 01010101 tag shows block free         */
#define USEDTAG 0xaa            /* 10101010 tag shows block used         */

static char allocbuf[ALLOCSIZE]; /* storage for allocation                */

struct blockl {                 /* info on left edge of block            */
   unsigned tag : 8;            /* left end tag: free or used            */
   unsigned size : 24;          /* length of block in bytes              */
   struct blockl *nextp;        /* pointer to next free block            */
   struct blockl *prevp;        /* pointer to previous free block        */
   };
struct blockr {                 /* info on right edge of block           */
   unsigned tag : 8;            /* right end tag: free or used           */
   unsigned size : 24;          /* length of block in bytes              */
   };
static struct blockl *freep;    /* head of free chain pointer            */
static struct blockl *cursorp;  /* cursor position in free chain         */
static void enchain(struct blockl *p);
static void unchain(struct blockl *p);
static void coalesce(struct blockl *blockl1p, struct blockl *blockl2p);
void dumpfree(void);

/* initalloc initializes all the structures for later alloc() calls       */
void initalloc()
{
  struct blockr * blockrp;      /* ptr to struct blockr at end of block   */

   cursorp = (freep = (struct blockl *) allocbuf);   /* set cursorp
      and freep to initial freeblock, which is whole buffer               */
   freep->tag = FREETAG;        /* tag it as free                        */
   freep->size = ALLOCSIZE;     /* initial size                          */
   freep->nextp = freep;        /* one block, points to self             */
   freep->prevp = freep;        /* ditto                                 */
    /* now point blockrp to last word of freeblock                       */
   blockrp = (struct blockr *)(allocbuf + ALLOCSIZE - TAGSIZE);
   blockrp->tag = FREETAG;      /* as above                              */
   blockrp->size = ALLOCSIZE;   /* as above                              */
}
```

```
static void enchain(struct blockl *p) /* place block on free chain         */
{
   struct blockl *holdp;        /* temporary holder                        */

   holdp = freep;               /* remember old block at head of chain     */
   freep = p;                   /* place new block on head of chain        */
    /* update structures to place new block into chain                     */
   if (holdp == NULL)  {        /* free chain had become empty             */
     freep->nextp = freep;      /* one block, points to self               */
     freep->prevp = freep;      /* ditto                                   */
     cursorp = freep;           /* and cursor pts to this blk              */
   }
   else  {                      /* old free chain non-empty                */
     /* insert on nextp list-- */
     freep->nextp = holdp;      /* new block points to old block           */
     holdp->prevp->nextp = freep;  /* end block points to new block        */
                                /* and insert on prevp list--              */
     freep->prevp = holdp->prevp;  /* new block points to end block        */
     holdp->prevp = freep;      /* old block points to new block           */
   }
}

static void unchain(struct blockl *p)
{
   if (p->prevp == p)           /* prevp points to self?                   */
      freep = (cursorp = NULL); /* empty list results                      */
   else  {                      /* chain around p->structl */
     (p->prevp)->nextp = p->nextp;
     (p->nextp)->prevp = p->prevp;
     if (p == cursorp)          /* cursorp was pting to this block?        */
       cursorp = p->nextp;      /* reset to next free block                */
     if (p == freep)            /* did we give away *freep ?               */
       freep = cursorp;         /* point freep to a free block             */
   }
}

/* alloc(): find a block of n bytes. Actually, find what might be somewhat
   more, allocxize >= n bytes, guaranteed to be longword-aligned, that is,
   starting on an address which is a multiple of 4, and is at least enough
   to make up a MINFREESIZE block, long enough to enchain when free. Note
   that a starting address multiple of 2 would be workable for numeric data
   on many machines (68000/68020), but a starting address multiple of 4
   for 32-bit numeric data has faster access characteristics on the 68020  */

char *alloc(int n)              /* return ptr to block of n characters     */
{
   struct blockr * blockrp;     /* ptr to struct blockr at end of block    */
   struct blockl *startp = cursorp,     /* search start in free chain      */
                 *holdp;        /* another ptr for general use             */
   int allocsize = 4*((n-1)/4+1)+2*TAGSIZE; /* size of block mult of 4     */
     /* note - always a multiple of 4, overhead of 2 tag/size blocks       */
   int newfreesize;             /* may leave a smaller free block behind   */
   char *p;                     /* for counting off chars in allocbuf      */

   /* look for a first fit for needed allocsize in free chain              */

   if (cursorp == NULL)         /* empty free chain?                       */
      return NULL;              /* failure to allocate                     */
```

```
   /* first make sure allocsize is large enough to enchain when freed      */

   allocsize = (allocsize > MINFREESIZE)? allocsize: MINFREESIZE;
   /* search for first fit                                                 */

   while (cursorp->size < allocsize   /* not large enough                  */
     && (cursorp = cursorp->nextp) != startp)   /* more to check?          */
         ;                            /* try again                        */
   if (cursorp->size < allocsize)     /* nothing large enough             */
      return(NULL);             /* failure to allocate                    */
   if (cursorp->size < allocsize)     /* nothing large enough             */
      return(NULL);             /* failure to allocate                    */

   /* found block large enough; can we carve off what we need and
      leave a freeblock?                                                   */
   if (cursorp->size >= allocsize + MINFREESIZE)  {  /* yes               */
      newfreesize = cursorp->size - allocsize;  /* size we will leave     */
      cursorp->size = newfreesize;  /* correct length of free block       */

     /* we will carve off the area to return from right end of free
        block we're on, so correct chaining for free block we're leaving
        already exists -- now fix up new left and right end structures    */

      p = (char *) cursorp;     /* need char pointer to count off bytes   */
      holdp = (struct blockl *) (p+=newfreesize); /* block to return      */

      blockrp = (struct blockr *) (p-TAGSIZE);  /* right end of free block */
      blockrp->size = newfreesize;   /* set up righthand free blocksize   */
      blockrp->tag = FREETAG;   /* set up righthand free tag              */

      holdp->tag = USEDTAG;     /* set up lefthand used tag               */
      holdp->size = allocsize;  /* allocated block size                   */
      blockrp = (struct blockr *) (p+allocsize-TAGSIZE); /* pt to right end */
      blockrp->tag = USEDTAG;   /* set tag                                */
      blockrp->size = allocsize;    /set size on right end                */
      return (((char *) holdp) + TAGSIZE); /* return char ptr AFTER tag/size*/
   }
   else   {                     /* won't leave a free block behind        */
      p = (char *) (holdp = cursorp); /* char pointer to current block    */
      holdp->tag = USEDTAG;     /* setr lefthand tag is used              */
      unchain(holdp);           /* take it out of the free chain          */
      blockrp = (struct blockr *) (p+(holdp->size)-TAGSIZE); /* right end  */
      blockrp->tag = USEDTAG;   /* set tag only for used block            */
      return(p+TAGSIZE);        /* char ptr AFTER tag/size                */
   }
}

/* free block with user-pointer p (block actually starts at p - TAGSIZE)   */
void freef(char *p)
{
   printf("freef called with p=%x\n",p);
}



 /* coalesce two free blocks, left on free list at call, right one not     */
```

```
void coalesce(struct blockl *blockl1p, struct blockl *blockl2p)
{
   struct blockr * blockrp;     /* ptr to struct blockr at end of block    */

   blockl1p->size += blockl2p->size;  /* new size is sum of old two        */
    /* now point to right end of enlarged new block                         */
   blockrp = (struct blockr *) (((char *) blockl1p)+blockl1p->size-TAGSIZE);
   blockrp->size = blockl1p->size;    /* set right end size                */
   return;
}

void dumpfree(void)
{
   struct blockl *p = freep;

   if (freep==NULL) printf("empty\n");
   else
     do
       {
         if (p==cursorp)
           printf("cursor-> ");
         else
           printf("         ");
         printf("freeblk of size %d at %x (%d from start)\n",p->size,p,
                ((char *)p-allocbuf));
       } while ((p=p->nextp)!=freep);
 }
```

```
/*********************************************************************
 *
 *                             alloctest
 *           interactive driver for storage allocator
 *
 *  Ethan Bolker CS240
 *  Fall 1988
 *  edited by Betty O'Neil Spring 1989, Spring 1990, Spring 2000
 *
 *********************************************************************/

/*
 * commands are
 *
 *    #      comment line
 *    a n  # call p = alloc(n), print p and save it in new slot in holdp
 *    f n  # call freef(n), where n is slot# of saved ptr in holdp
 *    d    # debug dump of free list (temp service, removed at end)
 */

#include <stdio.h>
#include <stdlib.h>
#include "alloc.h"

#define MAXSTR 100
#define ALLOC 'a'
#define FREE  'f'
#define DEBUGDUMP 'd'
#define COM   '#'
#define QUIT  'q'

char prompt[]= "> ";

void instruct(void);
int main(void);
/*********************************************************************/

int main()
{

  char line[MAXSTR];
  char command;
  int n;
  char * holdp[100];
  int holdindex=0;            /* in holdp */
  int done=0;

  instruct();
  initalloc();
  printf(prompt);
  while (!done && fgets( line, MAXSTR, stdin) != NULL) {
    command = line[0];
    switch (command) {
    case ALLOC: n = atoi(line+1);
      holdp[holdindex++] = alloc(n);
      printf("alloc #%d returned pointer: %p\n",
           holdindex-1,holdp[holdindex-1]);
      break;
```

```
    case FREE: n = atoi(line+1);
      freef(holdp[n]);
      break;
    case DEBUGDUMP: dumpfree();
      break;
    case COM:   break;
    case QUIT:  done=1;
      break;
    default:
      fprintf(stderr, "unknown command -%c-\n", command);
    }
    if (!done) printf(prompt);
  }
  putchar('\n');
  return(0);
}

void instruct()
{
  printf("interactive driver for testing storage allocator - commands are\n");
  printf("\n");
  printf("   #      comment line\n");
  printf("   a n  # call p = alloc(n), print p and save in new slot in
holdp\n");
  printf("   f n  # call freef(holdp[n]), i.e. free nth-obtained block\n");
  printf("   d    # dump free list (temp debugging aid)\n");
  printf("   q    # quit\n");
  printf("\n");
}
================================
TEST0 INPUT

# first allocate 10 blocks needing 100 bytes each incl. overhead--
a 92 # no. 0
a 92
a 92
a 92
a 92
a 92 # no. 5
a 92
a 92
a 92
d
a 92 # no. 9, i.e., 10th in all--exhausts 1000 bytes
a 8  # expect failure
f 1
d
f 3
d
a 48  # more than half of free block
a 48  # from other free block
a 36  # should be avail from leftover
a 36
f 5
f 7
f 6   # has free neighbors right and left
a 250 # should be a block this size to hand out
```