```c
#ifndef _KSYS_ALLOC
#include <stdio.h>
#include <string.h>
#include "kalloc.h"

/* The whole thing is: ("@" for the kheader_t of the block, "-" for free
 * memory, and "+" for allocated memory. One char for one unit.)
 *
 *           This region is core 1.                             This region is core 2.
 *
 *   @-------@++++++@++++++++++++@------------           @----------@++++++++++++@+++++++@------------
 *   |                           |                       |                               |
 *   p=p->ptr->ptr->ptr->ptr     p->ptr             p->ptr->ptr                p->ptr->ptr->ptr
 */

typedef struct _kheader_t_ {
    struct _kheader_t_ *ptr; /* next free block */
    size_t size; /* size of current free block */
} kheader_t;

typedef struct _allocated_t {
    struct _allocated_t *next;
    kheader_t *ptr;
} allocated_t;

static kheader_t base;
static kheader_t *loop_head = 0; /* the last and also the first free block */
static allocated_t kr_list_head, *kr_list_tail;
static size_t kr_total_allocated = 0;
static int kr_pow2 = 1;

void kr_set_pow2(int is_pow2)
{
    kr_pow2 = is_pow2;
}

static void kerror(const char *s)
{
    fprintf(stderr, "%s\n", s);
    exit(1);
}

static kheader_t *morecore(size_t nu)
{
    extern void kr_stat();
    extern void kfree(void*);
    size_t rnu;
    kheader_t *up;

    rnu = (nu + 0xffff) & (~(size_t)(0xffff));
    up = (kheader_t*)malloc(rnu * sizeof(kheader_t));
    if (!up) { /* fail to allocate memory */
        kr_stat();
        fprintf(stderr, "[morecore] %lu bytes requested but not available.\n", rnu * sizeof(kheader_t));
        exit(1);
    }
    /* put the pointer in kr_list_head */
    if (kr_list_tail == 0) kr_list_tail = &kr_list_head;
    kr_list_tail->ptr = up;
    kr_list_tail->next = (allocated_t*)calloc(1, sizeof(allocated_t));
    kr_list_tail = kr_list_tail->next;

    kr_total_allocated += rnu * sizeof(kheader_t);
    up->size = rnu; /* the size of the current block, and in this case the block is the same as the new core */
    kfree(up + 1); /* initialize the new "core" */
    return loop_head;
}

void kr_destroy()
{
    allocated_t *p, *q;
    p = &kr_list_head;
    do {
        q = p->next;
        free(p->ptr);
        if (p != &kr_list_head) free(p);
        p = q;
    } while (p->next);
    if (p != &kr_list_head) free(p);
}

void kfree(void *ap)
{
    kheader_t *p, *q;

    if (!ap) return;
    p = (kheader_t*)ap - 1; /* p->size is the size of the current block */
    /* Find the pointer that points to the block to be freed. The following loop can stop on two conditions:
     *
     * a) "p>q && p<q->ptr": @------@++++++++@+++++++@-------    @---------------@+++++++@-------
     *    (can also be in    |      |                |        -> |                       |
     *     two cores)        q      p           q->ptr           q                  q->ptr
     *
     *                       @--------    @+++++++++@--------    @--------    @------------------
     *                       |            |         |         -> |            |
     *                       q            p    q->ptr            q       q->ptr
     *
     * b) "q>=q->ptr && (p>q || p<q->ptr)":  @-------@+++++   @--------@+++++++     @-------@+++++   @----------------
     *                                       |                |        |         -> |               |
     *                                  q->ptr                q        p       q->ptr               q
     *
     *                                       @+++++++@-----   @++++++++@-------     @-------------   @++++++++@-------
     *                                       |       |                 |         -> |                        |
     *                                       p  q->ptr                 q       q->ptr                        q
     */
    for (q = loop_head; !(p > q && p < q->ptr); q = q->ptr)
        if (q >= q->ptr && (p > q || p < q->ptr)) break;
    if (p + p->size == q->ptr) { /* two adjacent blocks, merge p and q->ptr (the 2nd and 4th cases) */
        p->size += q->ptr->size; /* this is the new q->ptr size */
        p->ptr = q->ptr->ptr; /* this is the new q->ptr->ptr */
        /* p is actually the new q->ptr. The actual change happens a few lines below. */
    } else if (p + p->size > q->ptr && q->ptr >= p) { /* the end of the allocated block is in the next free block */
        kerror("[kfree] The end of the allocated block enters a free block.");
    } else p->ptr = q->ptr; /* backup q->ptr */

    if (q + q->size == p) { /* two adjacent blocks, merge q and p (the other two cases) */
        q->size += p->size;
        q->ptr = p->ptr;
        loop_head = q;
    } else if (q + q->size > p && p >= q) { /* the end of a free block in the allocated block */
        kerror("[kfree] The end of a free block enters the allocated block.");
    } else loop_head = q->ptr = p; /* in two cores, cannot be merged */
}

#define round_up(v) (--(v), (v)|=(v)>>1, (v)|=(v)>>2, (v)|=(v)>>4, (v)|=(v)>>8, (v)|=(v)>>16, (v)|=(v)>>32, ++(v))

void *krealloc(void *ap, size_t n_bytes)
{
    extern void *kmalloc(size_t);
    kheader_t *p, *q;
    size_t n_units;

    if (!ap) return kmalloc(n_bytes);
    n_units = 1 + (n_bytes + sizeof(kheader_t) - 1) / sizeof(kheader_t);
    p = (kheader_t*)ap - 1;
    if (p->size >= n_units) return ap;
    q = (kheader_t*)kmalloc(n_bytes);
    if (q != ap) memcpy(q, ap, (p->size  - 1) * sizeof(kheader_t));
    kfree(ap);
    return q;
}

void *kmalloc(size_t n_bytes)
{
    kheader_t *p, *q;
    size_t n_units;

    /* "n_units" means the number of units. The size of one unit equals to sizeof(kheader_t).
     * "1" is the kheader_t of a block, which is always required. */
    n_units = 1 + (n_bytes + sizeof(kheader_t) - 1) / sizeof(kheader_t);
    if (kr_pow2) round_up(n_units);

    if (!(q = loop_head)) { /* the first time when kmalloc() is called, intialization */
        base.ptr = loop_head = q = &base;
        base.size = 0;
    }
    for (p = q->ptr;; q = p, p = p->ptr) { /* search for a suitable block */
        if (p->size >= n_units) { /* p->size if the size of current block. This line means the current block is large enough. */
            if (p->size == n_units) q->ptr = p->ptr; /* no need to split the block */
            else { /* split the block */
                /* memory is allocated at the end of the block */
                p->size -= n_units; /* reduce the size of the free block */
                p += p->size; /* skip to the kheader_t of the allocated block */
                p->size = n_units; /* set the size */
            }
            loop_head = q; /* set the end of chain */
            return p + 1; /* skip the kheader_t */
        }
        if (p == loop_head) /* then ask for more "cores" */
```

```c
			if (!(p = morecore(n_units))) return 0; /* p==0 if fail to allocate, but morecore() will call exit(1) first */
	}
}

void kr_stat()
{
	unsigned n_blocks, n_units;
	size_t max_block = 0;
	kheader_t *p, *q;
	float frag;

	if (!(p = loop_head)) return;
	n_blocks = n_units = 0;
	do {
		q = p->ptr;
		if (p->size > max_block) max_block = p->size;
		n_units += p->size;
		if (p + p->size > q && q > p)
			kerror("[kr_stat] The end of a free block enters another free block.");
		p = q;
		++n_blocks;
	} while (p != loop_head);

	--n_blocks;
	frag = 1.0/1024.0 * n_units * sizeof(kheader_t) / n_blocks;
	fprintf(stderr, "[kr_stat] tot=%lu, free=%lu, n_block=%u, max_block=%lu, frag_len=%.3fK\n",
			kr_total_allocated, n_units * sizeof(kheader_t), n_blocks, max_block * sizeof(kheader_t), frag);
}

#endif /* _KSYS_ALLOC */

#ifdef _KTEST
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "kalloc.h"

int main()
{
	char **p;
	int i, j, k, n, m, N, do_alloc;
	n = 20000; N = 40000; m = 1024;
	srand48(time(0));
	p = (char**)kmalloc(sizeof(char*) * N);
	kr_set_pow2(1);
	kr_stat();
	for (i = 0; i < N; ++i) p[i] = 0;
	for (i = j = 0; i < N; ++i) {
		do_alloc = (drand48() < 1.0 - (double)j/n)? 1 : 0;
		if (j == 0) do_alloc = 1;
		else if (j == n) do_alloc = 0;
		if (do_alloc == 1) {
			if (drand48() > 0.5) {
				p[j++] = (char*)kmalloc(sizeof(char) * (int)(m * drand48() + 1.5));
			} else {
				k = (int)(drand48() * j);
				p[k] = (char*)krealloc(p[k], sizeof(char) * (int)(m * (1.0 + drand48()) + 0.5));
				if (k == j) ++j;
			}
		} else if (do_alloc == 0) {
			k = (int)(drand48() * j);
			kfree(p[k]); p[k] = 0;
		}
		if (i != 0 && i % 1000 == 0) kr_stat();
	}
	fprintf(stderr, "%lu\n", kr_size(p));
	for (i = 0; i < N; ++i) kfree(p[i]);
	kfree(p);
	kr_stat();
	/*
	k = kr_total_allocated / 16;
	for (i = 0; i < k; ++i) kmalloc(4);
	kr_stat();
	*/
	kr_destroy();
	return 0;
}
#endif
```