```
/*****************************************************************************
Garbage collector
*****************************************************************************/

#include <stdio.h>
int main()
{
typedef struct header {
   unsigned int    size;
   struct header   *next;
} header_t;
static header_t base;          /* Zero sized block to get us started. */
static header_t *freep = &base; /* Points to first free block of memory. */
static header_t *usedp;         /* Points to first used block of memory. */

/*
 * Scan the free list and look for a place to put the block. Basically, we're
 * looking for any block that the to-be-freed block might have been partitioned from.
 */
static void
add_to_free_list(header_t *bp)
{
   header_t *p;

   for (p = freep; !(bp > p && bp < p->next); p = p->next)
      if (p >= p->next && (bp > p || bp < p->next))
         break;

   if (bp + bp->size == p->next) {
      bp->size += p->next->size;
      bp->next = p->next->next;
   } else
      bp->next = p->next;

   if (p + p->size == bp) {
      p->size += bp->size;
      p->next = bp->next;
   } else
      p->next = bp;

   freep = p;
}

#define MIN_ALLOC_SIZE 4096 /* We allocate blocks in page sized chunks. */
```

```
/*
 * Request more memory from the kernel.
 */
static header_t *
morecore(size_t num_units)
{
    void *vp;
    header_t *up;

    if (num_units > MIN_ALLOC_SIZE)
        num_units = MIN_ALLOC_SIZE / sizeof(header_t);

    if ((vp = sbrk(num_units * sizeof(header_t))) == (void *) -1)
        return NULL;

    up = (header_t *) vp;
    up->size = num_units;
    add_to_free_list (up);
    return freep;
}
/*
 * Find a chunk from the free list and put it in the used list.
 */
void *
GC_malloc(size_t alloc_size)
{
    size_t num_units;
    header_t *p, *prevp;

    num_units = (alloc_size + sizeof(header_t) - 1) / sizeof(header_t) + 1;
    prevp = freep;

    for (p = prevp->next;; prevp = p, p = p->next) {
        if (p->size >= num_units) { /* Big enough. */
            if (p->size == num_units) /* Exact size. */
                prevp->next = p->next;
            else {
                p->size -= num_units;
                p += p->size;
                p->size = num_units;
            }

            freep = prevp;
```

```
            /* Add to p to the used list. */
            if (usedp == NULL)
                usedp = p->next = p;
            else {
                p->next = usedp->next;
                usedp->next = p;
            }

            return (void *) (p + 1);
        }
        if (p == freep) { /* Not enough memory. */
            p = morecore(num_units);
            if (p == NULL) /* Request for more memory failed. */
                return NULL;
        }
    }
}
#define UNTAG(p) (((unsigned int) (p)) & 0xfffffffc)

/*
 * Scan a region of memory and mark any items in the used list appropriately.
 * Both arguments should be word aligned.
 */
static void
scan_region(unsigned int *sp, unsigned int *end)
{
    header_t *bp;

    for (; sp < end; sp++) {
        unsigned int v = *sp;
        bp = usedp;
        do {
            if (bp + 1 <= v &&
                bp + 1 + bp->size > v) {
                    bp->next = ((unsigned int) bp->next) | 1;
                    break;
            }
        } while ((bp = UNTAG(bp->next)) != usedp);
    }
}
/*
 * Scan the marked blocks for references to other unmarked blocks.
 */
```

```
static void
scan_heap(void)
{
    unsigned int *vp;
    header_t *bp, *up;

    for (bp = UNTAG(usedp->next); bp != usedp; bp = UNTAG(bp->next)) {
        if (!((unsigned int)bp->next & 1))
            continue;
        for (vp = (unsigned int *)(bp + 1);
             vp < (bp + bp->size + 1);
             vp++) {
            unsigned int v = *vp;
            up = UNTAG(bp->next);
            do {
                if (up != bp &&
                    up + 1 <= v &&
                    up + 1 + up->size > v) {
                    up->next = ((unsigned int) up->next) | 1;
                    break;
                }
            } while ((up = UNTAG(up->next)) != bp);
        }
    }
}
/*
 * Find the absolute bottom of the stack and set stuff up.
 */
void
GC_init(void)
{
    static int initted;
    FILE *statfp;

    if (initted)
        return;

    initted = 1;

    statfp = fopen("/proc/self/stat", "r");
    assert(statfp != NULL);
    fscanf(statfp,
           "%*d %*s %*c %*d %*d %*d %*d %*d %*u "
           "%*lu %*lu %*lu %*lu %*lu %*lu %*ld %*ld "
```

```
        "%*ld %*ld %*ld %*ld %*llu %*lu %*ld "
        "%*lu %*lu %*lu %lu", &stack_bottom);
    fclose(statfp);

    usedp = NULL;
    base.next = freep = &base;
    base.size = 0;
}
/*
 * Mark blocks of memory in use and free the ones not in use.
 */
void
GC_collect(void)
{
    header_t *p, *prevp, *tp;
    unsigned long stack_top;
    extern char end, etext; /* Provided by the linker. */

    if (usedp == NULL)
        return;

    /* Scan the BSS and initialized data segments. */
    scan_region(&etext, &end);

    /* Scan the stack. */
    asm volatile ("movl %%ebp, %0" : "=r" (stack_top));
    scan_region(stack_top, stack_bottom);

    /* Mark from the heap. */
    scan_heap();

    /* And now we collect! */
    for (prevp = usedp, p = UNTAG(usedp->next);; prevp = p, p = UNTAG(p->next)) {
    next_chunk:
        if (!((unsigned int)p->next & 1)) {
            /*
             * The chunk hasn't been marked. Thus, it must be set free.
             */
            tp = p;
            p = UNTAG(p->next);
            add_to_free_list(tp);

            if (usedp == tp) {
                usedp = NULL;
```

```
            break;
        }

        prevp->next = (unsigned int)p | ((unsigned int) prevp->next & 1);
        goto next_chunk;
      }
      p->next = ((unsigned int) p->next) & ~1;
      if (p == usedp)
        break;
    }
}
}
```