CS 444 Project 4: The SLOB SLAB

Morgan Patch, Mark Bereza

Abstract

For this assignment, we implemented a best-fit algorithm for the Linux kernel's SLOB allocator and compared it's memory usage to the default first-fit implementation.

December 1, 2017

Contents

1 Design Plan

The distinction between first-fit and best-fit algorithms is simple: a first-fit algorithm will select the first block that is at least the size of the cache being allocated, where a best-fit algorithm will look at all blocks, and select (out of all blocks as large or larger than the cache) the one which is closest to the size of the cache.

For the SLOB allocator, the changes to the algorithm occur in two sections: in `slob_alloc()` for finding the best block and page, and in `slob_page_alloc()` for actually performing the allocation.

`slob_alloc()` now does the majority of the work. It keeps track of the best page, the index of the best block within that page, and the size of the best block within that page. It iterates over the entire list of pages; for each one, it then iterates over each block within the page. If it finds a block better than the current best block, it sets the best page to the current, the best block index to the current index, and the best block size to the size of the current block. If the current block is *exactly* the size of the requested allocation, it stops iterating immediately.

It then passes the best page, the best block index, and the size of the allocation to `slob_page_alloc()`, which returns to the index of the best block, then allocates within that block.

This is perhaps not the most elegant solution, as it requires iterating over every block every time a slab is allocated, but it is the simplest which meets the given requirements.

2 Questions

2.1 What do you think the main point of this assignment is?

I think that the main point of this assignment is for us to learn about how the Linux kernel implements SLOB memory allocation. This is particularly useful because the two more commonly-used memory allocation systems in the Linux kernel, SLAB and SLUB, are similar in principle, albeit more complex in implementation. Additionally, converting the existing first-fit algorithm implemented in slob.c to a best-fit algorithm forces us to consider how important issues like memory fragmentation are to memory performance.

2.2 How did you personally approach the problem? Design decisions, algorithm, etc.

The high level design of our best-fit SLOB implementation mostly follows the design plan found at the start of this paper. We did, however, diverge from the initial design plan in a few places for our final implementation:

1. First, instead of keeping track of the best total size of the block, the final implementation simply keeps track of the difference between the requested size and the actual size of current best block. This is more intuitive since the aim of the best-fit algorithm is to minimize this delta.

2. Second, the part of the algorithm that iterates over every block in a single page looking for the best fit was delegated to a helper function named slob_update_best(). This helper function would be called on each page searched by slob_alloc(). This function would take pointers to

the current page, the page containing the current best block, the index of the current best block within that page, and the current best delta as input parameters. Every time this helper method would find a block with a smaller delta than the current best, it would update these pointers with the new best block's information.

3. Third, slob_page_alloc() is only used by the best-fit algorithm when a new page must be allocated to accomodate the request. Otherwise, a new method named slob_direct_alloc() is called after the best block has been determined. This function takes pointers to the page and index for the best block and allocates directly to that block in a fashion similar to slob_page_alloc(). The reason for this is because slob_page_alloc() contains some first-fit logic that is unneeded when a fitting block has already been found by slob_update_best().

As for the code that would be used to compare the first-fit and best-fit algorithms, we decided to create a system call that would measure memory usage. We would then perform a series of large, differently-sized allocations and deallocations using the first-fit slob allocator and use the aforementioned system call to see memory statistics. Afterwards, we would perform the same allocations/deallocations using our own best-fit algorithm and compare the results.

2.3 How did you ensure your solution was correct? Testing details, for instance.

We tested whether the code was, in fact, finding the best fit by introducing printk statements which would print after a new "best" block was found (i.e. after the search found a block which was smaller than the previous best). These statements printed the size of the old and new "best" blocks, as well as the difference between the sizes and the requested allocation size. For example:

```
SLOB: Starting search...
SLOB: New best fit found:
        Size  = 1296
        Delta = 1268
SLOB: New best fit found:
        Size  = 248
        Delta = 220
SLOB: New best fit found:
        Size  = 180
        Delta = 152
SLOB: New best fit found:
        Size  = 168
        Delta = 140
SLOB: New best fit found:
        Size  = 144
        Delta = 116
SLOB: New best fit found:
        Size  = 136
        Delta = 108
SLOB: New best fit found:
        Size  = 128
        Delta = 100
SLOB: New best fit found:
        Size  = 96
        Delta = 68
SLOB: New best fit found:
        Size  = 92
        Delta = 64
SLOB: New best fit found:
        Size  = 40
        Delta = 12
SLOB: New best fit found:
        Size  = 36
        Delta = 8
SLOB: New best fit found:
        Size  = 32
        Delta = 4
SLOB: New best fit found:
        Size  = 28
        Delta = 0
SLOB: Ending search...
```

As long as the size and delta were strictly decreasing in a given search, while never going below the requested size/zero, this means that it was correctly searching for the best size.

2.4 What did you learn?

By completing this assignment, we the following:

1. First, we learned how the SLOB system fragments memory blocks when the requested size is smaller than the provided block and/or when the requsted memory must be n-byte aligned.
2. Second, we learned how to implement a new system call in the Linux kernel.
3. Third, we learned how to measure memory fragmentation using node and zone information.
4. Finally, we learned the difference between the first-fit and best-fit algorithms.

2.5 How should the TA test your patch?

First, apply our patch to a fresh copy of the 3.19.2 branch of Linux Yocto.

Grab the base config file, and run `make menuconfig`. Under General Setup, select the "Choose SLAB allocator" menu, then select the "SLOB (Simple Allocator)". Below that, turn on "Use SLOB best-fit" and "Print SLOB best-fit searches". The option "Print SLOB memory information" should be off at this point. Then, make the kernel with `make -j4 all`.

Run the kernel with the usual qemu command:

```
qemu-system-i386 -nographic -kernel arch/x86/boot/bzImage -drive file=
    core-image-lsb-sdk-qemux86.ext4,if=virtio -enable-kvm -net none -usb
    -localtime --no-reboot --append "root=/dev/vda rw console=ttyS0
    debug"
```

For starters, the simple fact that you will reach the login page shows that our memory management code is working and did not break the kernel.

You should also see output similar to that in question 2.3, and can verify that for each search, the difference between the requested size and the size of the current best block (delta) continuously decreases. This shows that the algorithm is finding the best-fit.

Now, exit qemu, and re-run `make menuconfig`. Under General Setup, turn off "Print SLOB best-fit searches". Re-make the kernel (this should be relatively quick).

Now, run a modified qemu command (it is the same as above, except that it opens a port for SSH):

```
qemu-system-i386 -nographic -kernel arch/x86/boot/bzImage -drive file=
    core-image-lsb-sdk-qemux86.ext4,if=virtio -enable-kvm -net user,
    hostfwd=tcp::5540-:22 -net nic -usb -localtime --no-reboot --append
    "root=/dev/vda rw console=ttyS0 debug"
```

In a separate terminal, run `scp -P 5540 test.c root@localhost:/home/root/` to copy the test program onto the running kernel.

In the original terminal, compile the test program using `gcc -o test test.c` and then run it using `time ./test`, which will allocate and free large amounts of memory to heavily use our code, then print out memory information (the number of allocated pages and the number of free pages). Copy this into another window for storing.

Close the terminal, and run `make menuconfig` one last time. Under General Setup, turn off "Use SLOB best-fit" to use the original first-fit slob allocator. Then run qemu, log in (the test program should already still be there), and run the test program to get the same information for the first-fit.

Comparing the outputs of the test program on the two algorithms should demonstrate the efficiency of best-fit. Here is the output we obtained after running the test program using first-fit:

```
Memory size: 33669120; memory used: 8728494

real    0m0.002s
user    0m0.000s
sys   0m0.002s
```

And here is the output we obtain after running the test program using best-fit:

```
Memory size: 31576064; memory used: 9823209

real    0m0.019s
user    0m0.002s
sys   0m0.016s
```

For first-fit, about 3.86 bytes of memory are needed per byte used, whereas for best-fit this ratio shrinks to about 3.21, an improvement of almost 17%. On the other hand, the time needed to perform these allocations is much longer for best-fit, taking about 8 times longer. This is likely because of the inefficient way the best-fit algorithm was implemented, which causes it to search every single block of every single page for each allocation.

3 Version Control Log

acronym	**meaning**
V	`version`
tag	`git tag`
MF	Number of `modified files`.
AL	Number of `added lines`.
DL	Number of `deleted lines`.

V	**tag**	**date**	**commit message**	**MF**	**AL**	**DL**
1		2017-10-19	Add Linux Yocto at tag 3.19.2.	48441	19142068	0
2		2017-11-29	Add structure for writeup.	3	134	0
3		2017-11-29	Add finalized design plan to writeup.	2	24	0
4		2017-11-29	Add best vs. first config option.	1	10	0
5		2017-11-30	wrapped best-fit code in #ifdefs	1	124	0
6		2017-11-30	wrote best-fit algorithm into slob.c	1	80	57
7		2017-11-30	fixed compile errors	1	5	4
8		2017-11-30	first fix attempt	1	53	55
9		2017-11-30	no longer hangs	1	6	0
10		2017-11-30	Add debug prints.	1	51	8
11		2017-12-01	best-fit algorithm now works	1	82	46
12		2017-12-01	Decrease print frequency and stop crashing.	1	33	22
13		2017-12-01	added content to writeup	1	11	0
14		2017-12-01	Add memory syscalls.	3	28	5
15		2017-12-01	change to printing	1	33	26
16		2017-12-01	fixed merge conflicts	1	20	2
17		2017-12-01	more work on writeup	3	405	0
18		2017-12-01	Added config options for printing	19	53	13
19		2017-12-01	Add testing writeup.	12	184	0
20		2017-12-01	Add test file	0	33	0
21		2017-12-01	Fix typos in test.	2	1	0
22		2017-12-01	updates to writeup	47	122	187

4 Work Log

- Wednesday, 2017-11-29
  - Looked at assignment description.
  - Came up with design plan for best-fit algorithm.
  - Documented design plan in LaTeX.
- Thursday, 2017-11-30
  - Added option to configuration for using best-fit for SLOB.
  - Wrapped first-fit and best-fit code in #ifdefs.
  - Started writing best-fit algorithm code.

  - Started writing memory fragmentation measuring code.
  - Added some debug print statements.

- Friday, 2017-12-01
  - Added more printk statements.
  - Finished best-fit algorithm.
  - Finished code to test memory use/fragmentation.
  - Made new system call for memory use testing.
  - Added config options for printing kernel messages.
  - Created memory test program.
  - Created patch file.
  - Finished writeup.
  - Finished assignment.

Appendix 1: Source Code

slob.c

```
#include <linux/syscalls.h>

#define SLOB_PRINT_LOWER_THRESHOLD 13
#define SLOB_PRINT_UPPER_THRESHOLD 15

unsigned long mem_size = 0;
unsigned long mem_used = 0;

/*
 * Allocate a slob block within a given slob_page sp.
 */
static void *slob_page_alloc(struct page *sp, size_t size, int align)
{
        slob_t *prev, *cur, *aligned = NULL;
        int delta = 0, units = SLOB_UNITS(size);

        for (prev = NULL, cur = sp->freelist; ; prev = cur, cur = slob_next(cur)) {
                slobidx_t avail = slob_units(cur);

                if (align) {
                        aligned = (slob_t *)ALIGN((unsigned long)cur, align);
                        delta = aligned - cur;
                }
                if (avail >= units + delta) { /* room enough? */
                        slob_t *next;

                        if (delta) { /* need to fragment head to align? */
                                next = slob_next(cur);
                                set_slob(aligned, avail - delta, next);
                                set_slob(cur, delta, aligned);
                                prev = cur;
                                cur = aligned;
                                avail = slob_units(cur);
                        }
                        next = slob_next(cur);
                        if (avail == units) { /* exact fit? unlink. */
                                if (prev)
                                        set_slob(prev, slob_units(prev), next);
                                else
                                        sp->freelist = next;
                        } else { /* fragment */
                                if (prev)
                                        set_slob(prev, slob_units(prev), cur + units);
                                else
                                        sp->freelist = cur + units;
                                set_slob(cur + units, avail - units, next);
                        }

                        sp->units -= units;
                        if (!sp->units)
                                clear_slob_page_free(sp);

                        mem_used += units;
                        return cur;
                }
                if (slob_last(cur))
                        return NULL;
        }
}

SYSCALL_DEFINE0(mem_size) {
        return mem_size;
}

SYSCALL_DEFINE0(mem_used) {
        return mem_used;
}

#ifdef CONFIG_SLOB_PRINT
static void print_meminfo(int start) {
        struct sysinfo info;
        pg_data_t *node;
        struct zone *zone;
        struct zone *node_zones;

        if (start)
                printk(KERN_NOTICE "SLOB: Starting allocation.");
        else
                printk(KERN_NOTICE "SLOB: Allocation finished.");

        si_meminfo(&info);
        printk(KERN_NOTICE "SLOB: Available memory: %lu\n", info.freeram);

        printk(KERN_NOTICE "SLOB: Memory fragmentation:\n");
        node = first_online_pgdat();
        node_zones = node->node_zones;
        for (zone = node_zones; zone - node_zones < MAX_NR_ZONES; zone++) {
                int order;
                unsigned long flags;

                if (!populated_zone(zone))
                        continue;

                spin_lock_irqsave(&zone->lock, flags);
                printk(KERN_NOTICE "SLOB: Node %d, zone %8s ", node->node_id, zone->name);
                for (order = 0; order < MAX_ORDER; ++order)
                        printk("%4lu ", zone->free_area[order].nr_free);
                printk("\n");
                spin_unlock_irqrestore(&zone->lock, flags);
        }
}
#endif

#ifdef CONFIG_SLOB_BEST_FIT
static void *slob_direct_alloc(struct page *sp,
                               slob_t *block,
                               slob_t *prev,
                               size_t size,
                               int align)
{
        slob_t *next, *aligned = NULL;
        int delta = 0, units = SLOB_UNITS(size);
        slobidx_t avail = slob_units(block);

        if (align) {
                aligned = (slob_t *)ALIGN((unsigned long)block, align);
                delta = aligned - block;
        }

        if (delta) { /* need to fragment head to align? */
                next = slob_next(block);
                set_slob(aligned, avail - delta, next);
                set_slob(block, delta, aligned);
                prev = block;
                block = aligned;
                avail = slob_units(block);
        }

        next = slob_next(block);
        if (avail == units) { /* exact fit? unlink. */
                if (prev)
                        set_slob(prev, slob_units(prev), next);
                else
                        sp->freelist = next;
        } else { /* fragment */
                if (prev)
                        set_slob(prev, slob_units(prev), block + units);
                else
                        sp->freelist = block + units;
                set_slob(block + units, avail - units, next);
        }

        sp->units -= units;
        mem_used += units;
        if (!sp->units)
                clear_slob_page_free(sp);
        return block;
}

/*
 * See if any of the slobs in a page are better than the current best.
 * If a slob is found of the exact size, return 1. Otherwise, return 0.
 */
#ifdef CONFIG_PRINT_BEST_FIT
static int slob_update_best(struct page *sp,
                            size_t size,
                            int align,
                            struct page **best_page,
                            slob_t **best_idx,
                            slob_t **best_idx_prev,
                            size_t *best_delta,
                            int *best_ctr,
                                slobidx_t avails[SLOB_PRINT_UPPER_THRESHOLD],
                                size_t deltas[SLOB_PRINT_UPPER_THRESHOLD])
{
#else
static int slob_update_best(struct page *sp,
                                size_t size,
                                int align,
                                struct page **best_page,
                                slob_t **best_idx,
                                slob_t **best_idx_prev,
                                size_t *best_delta)
{
#endif
        slob_t *prev, *cur, *aligned = NULL;
        int delta = 0, units = SLOB_UNITS(size), ret = 0;

        for (prev = NULL, cur = sp->freelist; ;
             prev = cur, cur = slob_next(cur)) {
                slobidx_t avail = slob_units(cur);
                int real_size = 0;

                if (align) {
                        aligned = (slob_t *)ALIGN((unsigned long)cur, align);
                        delta = aligned - cur;
                }

                real_size = units + delta;

                if ((avail >= real_size) &&
                    ((*best_page == NULL) ||
                     ((avail - real_size) < *best_delta))) {
                        *best_page      = sp;
                        *best_idx       = cur;
                        *best_idx_prev = prev;
                        *best_delta     = avail - real_size;
#ifdef CONFIG_PRINT_BEST_FIT
                        if (*best_ctr < SLOB_PRINT_UPPER_THRESHOLD) {
                                avails[(*best_ctr)] = avail;
                                deltas[(*best_ctr)] = *best_delta;
                        }
                        if (*best_ctr < 100)
                                (*best_ctr)++;
#endif
                        if (*best_delta == 0) {
                                ret = 1;
                                break;
                        }
                }
                if (slob_last(cur))
                        break;
        }

        return ret;
}

/*
 * slob_alloc: entry point into the slob allocator.
 */
#define PRINT_FREQ 8192
static void *slob_alloc(size_t size, gfp_t gfp, int align, int node)
{
#ifdef CONFIG_PRINT_BEST_FIT
        int i;
        int best_ctr = 0;
        slobidx_t avails[SLOB_PRINT_UPPER_THRESHOLD];
        size_t deltas[SLOB_PRINT_UPPER_THRESHOLD];
#endif
#ifdef CONFIG_SLOB_PRINT
        static unsigned long ctr = 0;
#endif
        struct page *sp, *best_page = NULL;
        struct list_head *slob_list;
        slob_t *b = NULL, *best_idx = NULL, *best_idx_prev = NULL;
        unsigned long flags;
        size_t best_delta = 0;

        if (size < SLOB_BREAK1)
                slob_list = &free_slob_small;
        else if (size < SLOB_BREAK2)
                slob_list = &free_slob_medium;
        else
                slob_list = &free_slob_large;
#ifdef CONFIG_SLOB_PRINT
        if (ctr & PRINT_FREQ)
                print_meminfo(true);
#endif
        spin_lock_irqsave(&slob_lock, flags);
        /* Iterate through each partially free page, try to find room */
        list_for_each_entry(sp, slob_list, lru) {
#ifdef CONFIG_NUMA
                /*
                 * If there's a node specification, search for a partial
                 * page with a matching node id in the freelist.
                 */
                if (node != NUMA_NO_NODE && page_to_nid(sp) != node)
                        continue;
#endif
                /* Enough room on this page? */
                if (sp->units < SLOB_UNITS(size))
                        continue;
#ifdef CONFIG_PRINT_BEST_FIT
                if (slob_update_best(sp, size, align, &best_page, &best_idx,
                                     &best_idx_prev, &best_delta, &best_ctr,
                                     avails, deltas))
                        break;
        }
        if ((best_ctr >= SLOB_PRINT_LOWER_THRESHOLD) &&
            (best_ctr <= SLOB_PRINT_UPPER_THRESHOLD)) {
                printk("SLOB: Starting search...\n");
                for (i = 0; i < best_ctr; ++i) {
                        printk("SLOB: New best fit found:\n");
                        printk("\tSize  = %d\n", avails[i]);
                        printk("\tDelta = %d\n", deltas[i]);
                }
                printk("SLOB: Ending search...\n");
#else
                if (slob_update_best(sp, size, align, &best_page, &best_idx,
                                     &best_idx_prev, &best_delta))
                        break;
#endif
        }
        if (best_page != NULL) {
                b = slob_direct_alloc(best_page, best_idx,
                                      best_idx_prev, size, align);
#ifdef CONFIG_SLOB_PRINT
                if (ctr & PRINT_FREQ)
                        printk("SLOB: best delta = %d\n", best_delta);
#endif
```

```
                spin_unlock_irqrestore(&slob_lock, flags);
        } else { /* Not enough space: must allocate a new page */
#ifdef CONFIG_SLOB_PRINT
                if (ctr & PRINT_FREQ)
                        printk("SLOB: Not enough space, allocating new page...\n");
#endif
                spin_unlock_irqrestore(&slob_lock, flags);
                b = slob_new_pages(gfp & ~__GFP_ZERO, 0, node);
                if (!b)
                        return NULL;
                sp = virt_to_page(b);
                __SetPageSlab(sp);
                spin_lock_irqsave(&slob_lock, flags);
                sp->units = SLOB_UNITS(PAGE_SIZE);
                sp->freelist = b;
                INIT_LIST_HEAD(&sp->lru);
                set_slob(b, SLOB_UNITS(PAGE_SIZE), b + SLOB_UNITS(PAGE_SIZE));
                set_slob_page_free(sp, slob_list);
                b = slob_page_alloc(sp, size, align);
                BUG_ON(!b);
                spin_unlock_irqrestore(&slob_lock, flags);
        }
        if (unlikely((gfp & __GFP_ZERO) && b))
                memset(b, 0, size);
#ifdef CONFIG_SLOB_PRINT
        if (ctr & PRINT_FREQ) {
                print_meminfo(false);
                ctr &= (PRINT_FREQ-1);
        }
        ctr++;
#endif
        return b;
}
#else
/*
 * slob_alloc: entry point into the slob allocator.
 */
#define PRINT_FREQ 8192
static void *slob_alloc(size_t size, gfp_t gfp, int align, int node)
{
        static unsigned long ctr = 0;
        struct page *sp;
        struct list_head *prev;
        struct list_head *slob_list;
        slob_t *b = NULL;
        unsigned long flags;

        if (size < SLOB_BREAK1)
                slob_list = &free_slob_small;
        else if (size < SLOB_BREAK2)
                slob_list = &free_slob_medium;
        else
                slob_list = &free_slob_large;
#ifdef CONFIG_SLOB_PRINT
        if (ctr & PRINT_FREQ)
                print_meminfo(true);
#endif
        spin_lock_irqsave(&slob_lock, flags);
        /* Iterate through each partially free page, try to find room */
        list_for_each_entry(sp, slob_list, lru) {
#ifdef CONFIG_NUMA
                /*
                 * If there's a node specification, search for a partial
                 * page with a matching node id in the freelist.
                 */
                if (node != NUMA_NO_NODE && page_to_nid(sp) != node)
                        continue;
#endif
                /* Enough room on this page? */
                if (sp->units < SLOB_UNITS(size))
                        continue;

                /* Attempt to alloc */
                prev = sp->lru.prev;
                b = slob_page_alloc(sp, size, align);
                if (!b)
                        continue;

                /* Improve fragment distribution and reduce our average
                 * search time by starting our next search here. (see
                 * Knuth vol 1, sec 2.5, pg 449) */
                if (prev != slob_list->prev &&
                                slob_list->next != prev->next)
                        list_move_tail(slob_list, prev->next);
                break;
        }
        spin_unlock_irqrestore(&slob_lock, flags);

        /* Not enough space: must allocate a new page */
        if (!b) {
                b = slob_new_pages(gfp & ~__GFP_ZERO, 0, node);
                if (!b)
                        return NULL;
                sp = virt_to_page(b);
                __SetPageSlab(sp);

                spin_lock_irqsave(&slob_lock, flags);
                sp->units = SLOB_UNITS(PAGE_SIZE);
                sp->freelist = b;
                INIT_LIST_HEAD(&sp->lru);
                set_slob(b, SLOB_UNITS(PAGE_SIZE), b + SLOB_UNITS(PAGE_SIZE));
                set_slob_page_free(sp, slob_list);
                b = slob_page_alloc(sp, size, align);
                BUG_ON(!b);
                spin_unlock_irqrestore(&slob_lock, flags);
        }
        if (unlikely((gfp & __GFP_ZERO) && b))
                memset(b, 0, size);
#ifdef CONFIG_SLOB_PRINT
        if (ctr & PRINT_FREQ) {
                printk("SLOB: requested = %d\n", size);
                printk("SLOB: actual    = %d\n", slob_units(b));
                print_meminfo(false);
                ctr &= (PRINT_FREQ-1);
        }
        ctr++;
#endif
        return b;
}
#endif
```

include/linux/syscalls.h

```
asmlinkage long sys_mem_size(void);
asmlinkage long sys_mem_used(void);
```

arch/x86/syscalls/syscall_32.tbl

```
359     i386    mem_size            sys_mem_size
360     i386    mem_used            sys_mem_used
```

test.c

```c
#include <stdio.h>
#include <sys/syscall.h>
#include <stdlib.h>

#define MEM_SIZE 359
#define MEM_USED 360

int main() {
        unsigned char *buf0, *buf1, *buf2, *buf3, *buf4, *buf5, *buf6, *buf7;

        buf0 = malloc(1234);
        buf1 = malloc(8192);
        free(buf0);
        buf2 = malloc(4096);
        buf3 = malloc(512);
        buf4 = malloc(4096);
        free(buf3);
        buf5 = malloc(256);
        buf6 = malloc(1048576);
        free(buf1);
        free(buf4);
        buf7 = malloc(2048);

        long size = (long)syscall(MEM_SIZE);
        long used = (long)syscall(MEM_USED);

        printf("Memory size: %l; memory used: %l\n", size, used);
        free(buf2);
        free(buf5);
        free(buf6);
        free(buf7);

        return 0;
}
```