دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

تمرین سری ششم - عملی

پارسا ملکیان - ۴۰۲۱۷۱۰۷۵
فاطمه شفیعی - ۴۰۲۱۱۰۸۸۷

سیستم‌های عامل
دکتر اسدی

۱ گزارش تمرین ۱ - پیاده‌سازی realloc وcalloc در xv۶

قبل از شروع پیاده‌سازی، ساختار موجود توابع malloc و free را بررسی کردیم. این پیاده‌سازی بر اساس الگوریتم Kernighan and Ritchie است.

ویژگی‌های کلیدی:

- استفاده از لیست پیوندی از بلوک‌های آزاد (free list)
- هر بلوک دارای یک header است که اندازه و اشاره‌گر به بلوک بعدی را نگه می‌دارد
- malloc: بلوک مناسب را در لیست آزاد جستجو می‌کند
- free: بلوک آزاد شده را به لیست آزاد اضافه می‌کند و در صورت امکان با بلوک‌های مجاور ادغام می‌کند

ساختار Header در umalloc.c

```
typedef union header {
  struct {
    union header *ptr;
    uint size;
  } s;
  Align x;
} Header;
```

۱.۱ پیاده‌سازی تابع **realloc**

۱.۱.۱ فایل: **user/umalloc.c**

تابع realloc به صورت زیر پیاده‌سازی شد:

```
void* realloc(void *ptr, uint nbytes)
```

۲.۱.۱ الگوریتم پیاده‌سازی

```
void*
realloc(void *ptr, uint nbytes)
{
  Header *bp;
  uint old_size;
  void *new_ptr;
  char *src, *dst;
  uint i;

  // If ptr is NULL, behave like malloc
  if(ptr == 0)
    return malloc(nbytes);
```

```
13
14    // If nbytes is 0, behave like free and return NULL
15    if(nbytes == 0){
16      free(ptr);
17      return 0;
18    }
19
20    // Get the header of the old block
21    bp = (Header*)ptr - 1;
22    old_size = (bp->s.size - 1) * sizeof(Header);
23
24    // Allocate new block
25    new_ptr = malloc(nbytes);
26    if(new_ptr == 0)
27      return 0;
28
29    // Copy old content to new block (copy minimum of old and new sizes)
30    src = (char*)ptr;
31    dst = (char*)new_ptr;
32    for(i = 0; i < old_size && i < nbytes; i++)
33      dst[i] = src[i];
34
35    // Free old block
36    free(ptr);
37
38    return new_ptr;
39  }
```

۱. بررسی **ptr == NULL**: اگر اشاره‌گر ورودی NULL باشد، تابع دقیقاً مانند malloc(nbytes) عمل می‌کند. این رفتار مطابق با استانداردهای C است.

۲. بررسی **nbytes == 0**: اگر اندازه درخواستی صفر باشد، حافظه آزاد می‌شود و NULL برگردانده می‌شود. این رفتار مانند free(ptr) است.

۳. تخصیص مجدد معمولی:

- ابتدا اندازه بلوک فعلی از header آن استخراج می‌شود.
- یک بلوک جدید با اندازه nbytes تخصیص داده می‌شود.
- محتوای قدیمی به بلوک جدید کپی می‌شود (حداقل اندازه قدیم و جدید).
- بلوک قدیمی آزاد می‌شود.
- اشاره‌گر به بلوک جدید برگردانده می‌شود.

۲.۱ پیاده‌سازی تابع calloc

۱.۲.۱ فایل: user/umalloc.c

تابع calloc به صورت زیر پیاده‌سازی شد:

```
void* calloc(uint num, uint size)
```

۲.۲.۱ الگوریتم پیاده‌سازی

```
void*
calloc(uint num, uint size)
{
  uint total_size;
  void *ptr;
  char *p;
  uint i;

  // Calculate total size needed
  total_size = num * size;

  // Allocate memory
  ptr = malloc(total_size);
  if(ptr == 0)
    return 0;

  // Initialize all bytes to zero
  p = (char*)ptr;
  for(i = 0; i < total_size; i++)
    p[i] = 0;

  return ptr;
}
```

۱. محاسبه اندازه کل: اندازه کل با ضرب تعداد عناصر (num) در اندازه هر عنصر (size) محاسبه می‌شود.

۲. تخصیص حافظه: با استفاده از malloc فضای مورد نیاز تخصیص داده می‌شود. در صورت شکست، NULL برگردانده می‌شود.

۳. مقداردهی اولیه: تمام بایت‌های فضای تخصیص داده شده به صفر مقداردهی می‌شوند. این تفاوت اصلی calloc با malloc است.

۳.۱ اضافه کردن تعریف توابع به فایل header

۱.۳.۱ فایل: user/user.h

برای اینکه برنامه‌های یوزر بتوانند از توابع جدید استفاده کنند، باید تعاریف آنها را به فایل header اضافه کنیم:

```
// umalloc.c
void* malloc(uint);
void free(void*);
void* realloc(void*, uint);
void* calloc(uint, uint);
```

قبل از تغییر:

```
// umalloc.c
void* malloc(uint);
void free(void*);
```

بعد از تغییر:

```
// umalloc.c
void* malloc(uint);
void free(void*);
void* realloc(void*, uint);
void* calloc(uint, uint);
```

۴.۱ ایجاد برنامه تست

۱.۴.۱ فایل: user/memtest.c

برای تست کامل توابع پیاده‌سازی شده، یک برنامه جامع نوشتیم که شامل چندین دسته تست است.

۲.۴.۱ تست‌های calloc

```
void test_calloc() {
    printf("\n=== Testing calloc ===\n");

    // Test 1: Basic calloc functionality
    printf("Test 1: Basic calloc - allocate 10 integers\n");
    int *arr = (int*)calloc(10, sizeof(int));

    // Check if all bytes are initialized to zero
    int all_zero = 1;
    for(int i = 0; i < 10; i++) {
        if(arr[i] != 0) {
            all_zero = 0;
            break;
        }
    }
    if(all_zero) {
        printf("  PASSED: All elements initialized to zero\n");
    }

    // Test 2: Write and read data
    for(int i = 0; i < 10; i++) {
        arr[i] = i * 2;
    }
    // Verify data...

    free(arr);
}
```

تست‌های انجام شده:

- تخصیص حافظه برای ۱۰ عدد صحیح
- بررسی مقداردهی اولیه به صفر
- نوشتن و خواندن داده‌ها
- تخصیص با اندازه‌های مختلف (۱۰۰ کاراکتر)

۳.۴.۱ تست‌های realloc

```
void test_realloc() {
    printf("\n=== Testing realloc ===\n");

    // Test 1: realloc with NULL pointer (should behave like malloc)
    printf("Test 1: realloc(NULL, 50) - should act like malloc\n");
    int *ptr = (int*)realloc(0, 50 * sizeof(int));

    // Write some data
    for(int i = 0; i < 50; i++) {
        ptr[i] = i;
    }

    // Test 2: Expand allocation
    printf("Test 2: Expand from 50 to 100 integers\n");
    int *new_ptr = (int*)realloc(ptr, 100 * sizeof(int));

    // Check if old data is preserved
    int data_preserved = 1;
    for(int i = 0; i < 50; i++) {
        if(new_ptr[i] != i) {
            data_preserved = 0;
            break;
        }
    }
    if(data_preserved) {
        printf("  PASSED: Old data preserved after expansion\n");
    }

    // Test 3: Shrink allocation
    // Test 4: realloc with nbytes = 0 (should free)
}
```

تست‌های انجام شده:

- realloc(NULL, size) ـ رفتار مانند malloc
- بزرگ کردن حافظه (۵۰ به ۱۰۰ عنصر)
- کوچک کردن حافظه (۱۰۰ به ۲۵ عنصر)
- realloc(ptr, 0) ـ رفتار مانند free
- حفظ داده‌ها در تمام حالات

۴.۴.۱ تست‌های ترکیبی

```
void test_combined() {
    printf("\n=== Testing Combined Scenarios ===\n");

    // Test 1: Use calloc, then realloc
    int *arr = (int*)calloc(5, sizeof(int));

    // Write data
    for(int i = 0; i < 5; i++) {
        arr[i] = 100 + i;
    }

    // Realloc to larger size
    arr = (int*)realloc(arr, 10 * sizeof(int));

    // Verify data preserved
    // Test 2: Multiple allocations
}
```

تست‌های انجام شده:

- استفاده ترکیبی از calloc و realloc
- چندین تخصیص همزمان
- آزاد سازی‌های متعدد

۵.۱ تست‌های حالات خاص (Edge Cases)

```
void test_edge_cases() {
    printf("\n=== Testing Edge Cases ===\n");

    // Test 1: Large allocation
    char *large = (char*)calloc(5000, sizeof(char));

    // Test 2: Zero elements
    void *zero_num = calloc(0, 10);

    // Test 3: Zero size
    void *zero_size = calloc(10, 0);
}
```

تست‌های انجام شده:

- تخصیص حافظه بزرگ (۵۰۰۰ بایت)
- تخصیص با تعداد صفر عنصر
- تخصیص با اندازه صفر

۶.۱ اضافه کردن برنامه تست به Makefile

۱.۶.۱ فایل: Makefile

برای اینکه برنامه تست در xv6 کامپایل و در دسترس باشد، باید آن را به لیست UPROGS اضافه کنیم.

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    # ... other programs ...
    $U/_lotterytest\
    $U/_memtest\
```

۷.۱ نتیجه اجرا

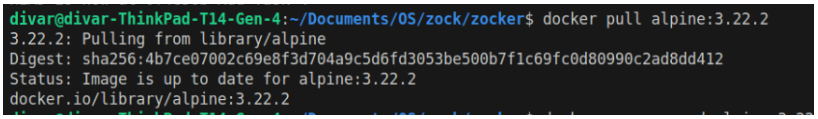
```
divar@divar-ThinkPad-T14-Gen-4:~/Documents/OS/6_n/Opperating-System-Practical/xv6-riscv$ make qemu CPUS=1
xv6 kernel is booting

init: starting sh
$ memtest
================================
Memory Allocation Test Suite
Testing realloc() and calloc()
================================

=== Testing calloc ===
Test 1: Basic calloc - allocate 10 integers
  PASSED: All elements initialized to zero
Test 2: Write and read data
  PASSED: Data written and read correctly
Test 3: calloc with 100 chars
  PASSED: All 100 bytes initialized to zero
calloc tests completed!

=== Testing realloc ===
Test 1: realloc(NULL, 50) - should act like malloc
  PASSED: Memory allocated
Test 2: Expand from 50 to 100 integers
  PASSED: Old data preserved after expansion
Test 3: Shrink from 100 to 25 integers
  PASSED: Data preserved after shrinking
Test 4: realloc(ptr, 0) - should act like free
  PASSED: Returned NULL and freed memory
realloc tests completed!

=== Testing Combined Scenarios ===
Test 1: calloc then realloc
  PASSED: calloc initialized to zero, realloc preserved data
Test 2: Multiple allocations and deallocations
  PASSED: Multiple allocations succeeded
Combined tests completed!

=== Testing Edge Cases ===
Test 1: Large allocation with calloc
  PASSED: Large allocation succeeded
Test 2: calloc(0, 10)
  INFO: Allocated memory for 0 elements
Test 3: calloc(10, 0)
  INFO: Allocated memory for 0-byte elements
Edge case tests completed!

================================
All tests completed!
================================
$
```

۲ گزارش تمرین ۲

۱.۲ اجرای کانتینر با ایمیج Alpine

یک کانتینر بر اساس ایمیج Alpine با دستور زیر اجرا کردیم:

```
docker run --rm -d alpine:3.22.2 sleep 1000
```



نتیجه: Container ID: 7e9a61c7ca6b3adaefd8cd8271531b6c73c0ef24fdb28cf2ed1aa6a0afc32b8a

توضیح پرچم **rm–**: این پرچم به Docker می‌گوید که به محض متوقف شدن کانتینر، آن را به صورت خودکار حذف کند. این کار باعث می‌شود که کانتینرهای متوقف شده در سیستم انباشته نشوند و فضای دیسک را اشغال نکنند.

۲.۲ Export کردن فایل سیستم کانتینر

با استفاده از دستور زیر، ساختار فایل سیستم کانتینر را در فایل export.tar ذخیره کردیم:

```
docker export 7e9a61c7ca6b -o export.tar
```

این دستور تمام فایل‌های موجود در فایل سیستم کانتینر (شامل /bin, /etc, /lib و غیره) را در یک فایل tar ذخیره می‌کند.

۳.۲ استخراج و مقایسه محتوای فایل

محتوای فایل export.tar را در دایرکتوری موقت استخراج کردیم:

```
mkdir -p /tmp/export_test
tar -xf export.tar -C /tmp/export_test
ls -la /tmp/export_test/
```

نتیجه ساختار دایرکتوری:

```
divar@divar-ThinkPad-T14-Gen-4:~/Documents/OS/zock/zocker$ docker export 7e9a61c7ca6b -o export.tar
divar@divar-ThinkPad-T14-Gen-4:~/Documents/OS/zock/zocker$ mkdir -p /tmp/export_test && tar -xf export.tar -C /tmp/export_test && ls -la /tmp/export_test/
total 84
drwxrwxr-x  19 divar divar  4096 Dec 23 07:21 .
drwxrwxrwt 100 root  root  12288 Dec 23 07:21 ..
drwxr-xr-x   2 divar divar  4096 Oct  8 12:58 bin
drwxr-xr-x   4 divar divar  4096 Dec 23 07:20 dev
-rwxr-xr-x   1 divar divar     0 Dec 23 07:20 .dockerenv
drwxr-xr-x  17 divar divar  4096 Dec 23 07:20 etc
drwxr-xr-x   2 divar divar  4096 Oct  8 12:58 home
drwxr-xr-x   6 divar divar  4096 Oct  8 12:58 lib
drwxr-xr-x   5 divar divar  4096 Oct  8 12:58 media
drwxr-xr-x   2 divar divar  4096 Oct  8 12:58 mnt
drwxr-xr-x   2 divar divar  4096 Oct  8 12:58 opt
dr-xr-xr-x   2 divar divar  4096 Oct  8 12:58 proc
drwx------   2 divar divar  4096 Oct  8 12:58 root
drwxr-xr-x   3 divar divar  4096 Oct  8 12:58 run
drwxr-xr-x   2 divar divar  4096 Oct  8 12:58 sbin
drwxr-xr-x   2 divar divar  4096 Oct  8 12:58 srv
drwxr-xr-x   2 divar divar  4096 Oct  8 12:58 sys
drwxrwxr-x   2 divar divar  4096 Oct  8 12:58 tmp
```

۱.۳.۲ مقایسه دایرکتوری /proc

در فایل export شده:

```
ls -la /tmp/export_test/proc/
```

```
divar@divar-ThinkPad-T14-Gen-4:~/Documents/OS/zock/zocker$ ls -la /tmp/export_test/proc/
total 8
dr-xr-xr-x  2 divar divar 4096 Oct  8 12:58 .
drwxrwxr-x 19 divar divar 4096 Dec 23 07:21 ..
```

در کانتینر در حال اجرا:

```
docker exec 7e9a61c7ca6b ls -la /proc/ | head -20
```

```
divar@divar-ThinkPad-T14-Gen-4:~/Documents/OS/zock/zocker$ docker exec 7e9a61c7ca6b ls -la /proc/ | head -20
total 4
dr-xr-xr-x  468 root     root            0 Dec 23 03:50 .
drwxr-xr-x    1 root     root         4096 Dec 23 03:50 ..
dr-xr-xr-x    9 root     root            0 Dec 23 03:50 1
dr-xr-xr-x    9 root     root            0 Dec 23 03:51 7
drwxrwxrwt    2 root     root           40 Dec 23 03:50 acpi
drwxrwxrwt    2 root     root           40 Dec 23 03:50 asound
-r--r--r--    1 root     root            0 Dec 23 03:51 bootconfig
-r--r--r--    1 root     root            0 Dec 23 03:51 buddyinfo
dr-xr-xr-x    4 root     root            0 Dec 23 03:50 bus
-r--r--r--    1 root     root            0 Dec 23 03:51 cgroups
-r--r--r--    1 root     root          119 Dec 23 03:51 cmdline
-r--r--r--    1 root     root            0 Dec 23 03:51 consoles
-r--r--r--    1 root     root            0 Dec 23 03:51 cpuinfo
-r--r--r--    1 root     root            0 Dec 23 03:51 crypto
-r--r--r--    1 root     root            0 Dec 23 03:51 devices
-r--r--r--    1 root     root            0 Dec 23 03:51 diskstats
-r--r--r--    1 root     root            0 Dec 23 03:51 dma
dr-xr-xr-x    4 root     root            0 Dec 23 03:51 driver
dr-xr-xr-x    3 root     root            0 Dec 23 03:51 dynamic_debug
```

علت تفاوت: دایرکتوری /proc یک filesystem مجازی (pseudo-filesystem) است که توسط کرنل لینوکس در زمان اجرا ایجاد می‌شود و اطلاعات پروسس‌ها و سیستم را نمایش می‌دهد. دستور docker export فقط فایل‌های واقعی را export می‌کند و های filesystem مجازی مانند /proc، /sys، و /dev را شامل نمی‌شود. در کانتینر در حال اجرا، /proc توسط کرنل mount شده و پر از اطلاعات است، اما در export شده فقط یک دایرکتوری خالی است.

۴.۲ استخراج در دایرکتوری نهایی

محتوای فایل را در مسیر /tmp/zocker/mycontainer استخراج کردیم:

```
mkdir -p /tmp/zocker/mycontainer
tar -xf export.tar -C /tmp/zocker/mycontainer
ls -la /tmp/zocker/mycontainer/
```

۵.۲ تغییرات کد

تابع run_container در فایل src/run.c را تغییر دادیم تا از پارامتر base_dir پشتیبانی کند.

۱.۵.۲ تغییرات اعمال شده

۱. بررسی وجود base_dir: اگر base_dir مقدار داشته باشد، از آن به عنوان دایرکتوری کانتینر استفاده می‌کنیم، در غیر این صورت از setup_container_dir استفاده می‌کنیم.

۲. مدیریت دایرکتوری **proc/**: چون در base_dir مستخرج شده، دایرکتوری /proc از قبل وجود دارد، قبل از ایجاد آن بررسی می‌کنیم.

کد قبل از تغییر

```
if (pid == 0) {
    char container_dir[256];
    if (setup_container_dir(cont.id, container_dir) != 0) {
      fprintf(stderr, "[ERR] Failed to setup container directory for %s\n",
              cont.id);
      return 1;
    }
    // ...
}
```

کد بعد از تغییر

```
if (pid == 0) {
    char container_dir[256];

    // Use base_dir if provided, otherwise setup container directory
    if (strlen(cont.base_dir) > 0) {
      strncpy(container_dir, cont.base_dir, sizeof(container_dir) - 1);
      container_dir[sizeof(container_dir) - 1] = '\0';
    } else {
      if (setup_container_dir(cont.id, container_dir) != 0) {
        fprintf(stderr, "[ERR] Failed to setup container directory for %s\n",
                cont.id);
        return 1;
      }
    }
    // ...
}
```

و همچنین برای مدیریت /proc:

```
// Create /proc directory if it doesn't exist (it might exist in base_dir)
struct stat st;
if (stat("/proc", &st) == -1) {
  if (mkdir("/proc", 0555) != 0) {
    fprintf(stderr, "[ERR] Failed to create /proc directory: %s\n",
            strerror(errno));
    return 1;
  }
}
```

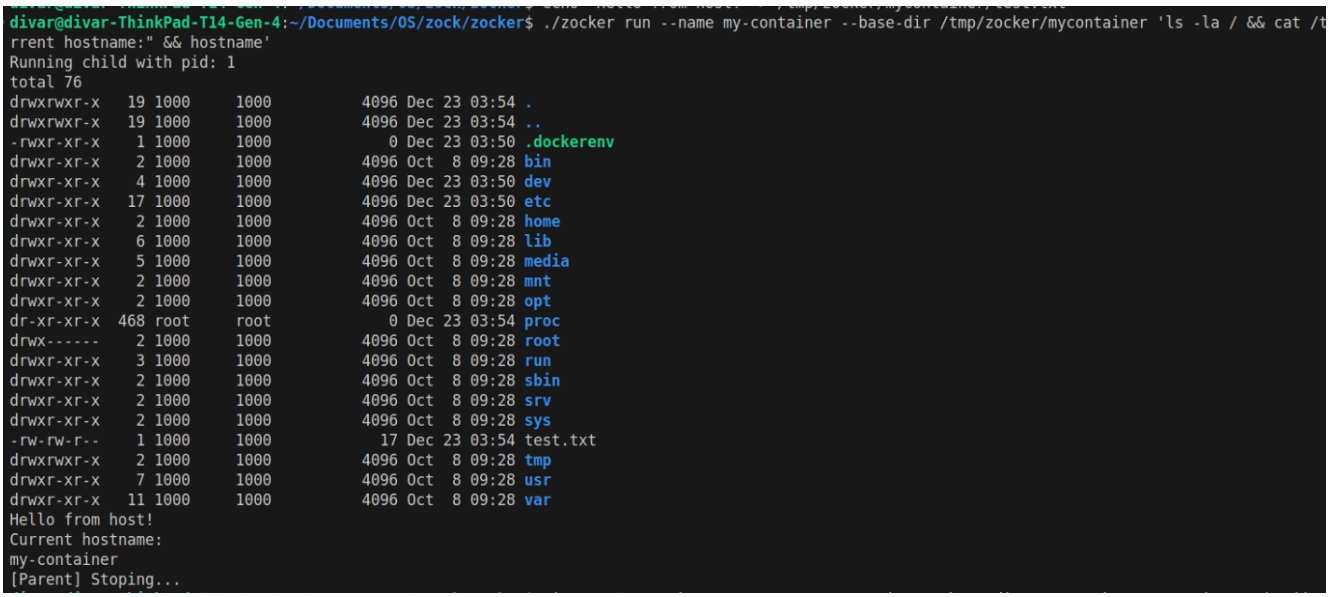
۶.۲ کامپایل و تست

پروژه را کامپایل کرده و تست کردیم:

```
make clean && make
```

```
./zocker run --name my-container --base-dir /tmp/zocker/mycontainer \
    'ls -la / && cat /test.txt && echo "Current hostname:" && hostname'
```

نتیجه:



موفقیت‌آمیز: کانتینر با موفقیت اجرا شد و قابلیت‌ها (دسترسی به فایل‌های base_dir، namespace PID جدا، hostname تنظیم‌شده، و /proc mount شده) برقرار بودند.

۷.۲ پاسخ به سوالات

۱.۷.۲ سوال ۱: پرچم rm– چه کاربردی دارد؟

پرچم rm– مشخص می‌کند کانتینر پس از توقف، به صورت خودکار حذف شود. مزایا:

- مدیریت خودکار منابع و جلوگیری از انباشته شدن کانتینرهای متوقف شده
- صرفه‌جویی در فضای دیسک
- تمیزکاری خودکار (عدم نیاز به docker rm)
- مناسب برای تست و کانتینرهای موقت

۲.۷.۲ سوال ۲: علت تفاوت محتوای پوشه /proc

مشاهده:

- در فایل export شده: /proc خالی است
- در کانتینر در حال اجرا: /proc پر از فایل‌ها و دایرکتوری‌هاست

علت:

۱. /proc یک pseudo-filesystem است که توسط کرنل در runtime ساخته می‌شود.
۲. docker export فقط فایل‌های واقعی را خروجی می‌گیرد و محتویات های filesystem مجازی را کپی نمی‌کند.
۳. در کانتینر اجرا شده، /proc با mount از نوع proc فعال و populate می‌شود.

۳.۷.۲ سوال ۳: آیا فایل اضافه شده به /tmp/zocker/mycontainer/test.txt از داخل کانتینر قابل دسترسی است؟

بله، کاملاً قابل دسترسی است.

۱. با chroot(/tmp/zocker/mycontainer)، دید پروسس کانتینر از root فایل سیستم تغییر می‌کند.
۲. فایل /tmp/zocker/mycontainer/test.txt روی میزبان، داخل کانتینر به شکل /test.txt دیده می‌شود.
۳. تغییرات در این مسیر روی میزبان، داخل کانتینر هم قابل مشاهده است.

```
# in host:
echo "Hello from host!" > /tmp/zocker/mycontainer/test.txt

# in container:
./zocker run --name my-container --base-dir /tmp/zocker/mycontainer 'cat /test.txt'
# result: Hello from host!
```

۴.۷.۲ سوال ۴: آیا در پیاده‌سازی اصلی Docker از همین روش استفاده شده است؟ اگر خیر، یکی از دلایل مهم ناکارآمد بودن این روش را ذکر کنید.

خیر. Docker از روش ساده کپی کامل فایل سیستم استفاده نمی‌کند و به جای آن از Filesystem Layered و مکانیزم Copy-on-Write استفاده می‌کند.

یکی از دلایل مهم ناکارآمدی: مصرف بالای فضای دیسک

اگر ۱۰۰ کانتینر با یک ایمیج یکسان داشته باشیم:

- روش کپی کامل: $100 \times 5MB = 500MB Docker : base layer +$

مثال:

```
1
 1: /tmp/zocker/container1/ (5 MB)
 2: /tmp/zocker/container2/ (5 MB)
...
 100: /tmp/zocker/container100/ (5 MB)
#   : 500 MB    100        !
```

راه‌حل Docker: filesystem union (مثل OverlayFS که در آن:

- لایه‌های پایه read-only و مشترک هستند
- هر کانتینر فقط یک layer قابل‌نوشتن برای تغییرات دارد

۳ گزارش تمرین ۳ - توسعه سیستم مدیریت فایل

۱.۳ مقدمه

در این تمرین، هدف تکمیل سامانه مدیریت فایل با پیاده‌سازی مکانیزم‌های امنیتی و سطوح دسترسی است. ویژگی‌های اضافه شده شامل مدیریت کاربران و گروه‌ها، و تعریف سطوح دسترسی برای فایل‌ها می‌باشد.

۲.۳ گام ۱: پیاده‌سازی ساختار داده‌های کاربران و گروه‌ها

۱.۲.۳ ساختارکاربر

در فایل filesystem.h ساختارکاربر تعریف شد:

```
typedef struct {
    int32_t uid;                        // User ID (0 = root)
    char username[MAX_USERNAME];
    int32_t primary_gid;                // Primary group ID
    int32_t groups[MAX_GROUPS_PER_USER]; // Secondary groups
    int32_t group_count;                // Number of secondary groups
    int32_t is_active;                  // 1 = active, 0 = deleted
} User;
```

۲.۲.۳ ساختارگروه

```
typedef struct {
    int32_t gid;                        // Group ID
    char groupname[MAX_GROUPNAME];
    int32_t is_active;                  // 1 = active, 0 = deleted
} Group;
```

۳.۲.۳ کاربر و گروه ریشه

هنگام فرمت دیسک جدید، کاربر root با uid=0 و گروه root با gid=0 به صورت خودکار ایجاد می‌شوند:

```
static void init_users_and_groups() {
    // Create root group (gid=0)
    group_table[0].gid = 0;
    strcpy(group_table[0].groupname, "root");
    group_table[0].is_active = 1;

    // Create root user (uid=0)
    user_table[0].uid = 0;
    strcpy(user_table[0].username, "root");
    user_table[0].primary_gid = 0;
    user_table[0].is_active = 1;
```

```
    current_uid = 0;  // Start as root
}
```

۳.۳ گام ۲: پیاده‌سازی دستورات مدیریت کاربران

۱.۳.۳ دستور useradd

```
myfs:root> useradd alice
 Group 'alice' created (gid=1)
 User 'alice' created (uid=1, gid=1)
```

این دستور یک کاربر جدید ایجاد می‌کند و به صورت خودکار یک گروه همنام برای آن می‌سازد.

۲.۳.۳ دستور userdel

```
myfs:root> userdel alice
  User 'alice' deleted
```

۳.۳.۳ دستور groupadd

```
myfs:root> groupadd developers
 Group 'developers' created (gid=2)
```

۴.۳.۳ دستور groupdel

```
myfs:root> groupdel developers
  Group 'developers' deleted
```

۵.۳.۳ دستور usermod -aG

این دستور کاربر را به یک گروه اضافه می‌کند:

```
myfs:root> usermod -aG alice developers
 User 'alice' added to group 'developers'
```

نکته: تمام دستورات مدیریت کاربران و گروه‌ها فقط توسط root قابل اجرا هستند.

۴.۳ گام ۳: اضافه کردن ویژگی‌های امنیتی به فایل‌ها

ساختار FileEntry با فیلدهای مالک و گروه گسترش یافت:

```
typedef struct {
    char name[MAX_FILENAME];
    int32_t size;
    int32_t start_block;
    int32_t block_count;
    uint32_t permissions;  // 0644, etc (owner:group:others)
    int32_t owner_uid;     // Owner user ID
    int32_t group_gid;     // Owner group ID
    time_t create_time;
    time_t modify_time;
    int32_t is_used;
} FileEntry;
```

هنگام ایجاد فایل، مالک و گروه به صورت خودکار تنظیم می‌شوند:

```
file_table[idx].owner_uid = current_uid;
file_table[idx].group_gid = user_table[current_uid].primary_gid;
```

۵.۳ گام ۴: پیاده‌سازی دستورات سطح دسترسی

۱.۵.۳ دستور chmod

تغییر سطح دسترسی فایل (فقط مالک یا root):

```
myfs:root> chmod 755 test.txt
 Permissions of 'test.txt' changed to 755
```

پیاده‌سازی:

```
int fs_chmod(const char* path, uint32_t mode) {
    int idx = find_file_index(path);
    if (idx < 0) return -1;

    // Only owner or root can chmod
    if (current_uid != 0 && file_table[idx].owner_uid != current_uid) {
        printf("Permission denied\n");
        return -1;
    }

    file_table[idx].permissions = mode;
    save_metadata();
    return 0;
}
```

۲.۵.۳ دستور chown

تغییر مالک فایل (فقط root):

```
myfs:root> chown alice:developers test.txt
 Owner of 'test.txt' changed to 'alice'
 Group of 'test.txt' changed to 'developers'
```

۳.۵.۳ دستور chgrp

تغییر گروه فایل (مالک یا root):

```
myfs:root> chgrp developers test.txt
 Group of 'test.txt' changed to 'developers'
```

۴.۵.۳ دستور getfacl

نمایش لیست کنترل دسترسی فایل:

```
myfs:root> getfacl test.txt

 Access Control List for 'test.txt':

  File:    test.txt
  Owner:   alice (uid=1)
  Group:   developers (gid=2)
  Mode:    755

  user::rwx
  group::r-x
  other::r-x
```

۶.۳ گام ۵: به‌روزرسانی دستورات قبلی با بررسی دسترسی

۱.۶.۳ تابع بررسی دسترسی

```
int fs_check_permission(const char* path, int required_perm) {
    int idx = find_file_index(path);
    if (idx < 0) return 0;

    // Root has all permissions
    if (current_uid == 0) return 1;

    FileEntry* file = &file_table[idx];
    uint32_t perm = file->permissions;

    int effective_perm;
```

```
    if (file->owner_uid == current_uid) {
        effective_perm = (perm >> 6) & 0x7;  // Owner bits
    }
    else if (fs_user_in_group(current_uid, file->group_gid)) {
        effective_perm = (perm >> 3) & 0x7;  // Group bits
    }
    else {
        effective_perm = perm & 0x7;  // Others bits
    }

    return (effective_perm & required_perm) == required_perm;
}
```

۲.۶.۳ به‌روزرسانی دستور open

```
void cmd_open(int argc, char args[][MAX_COMMAND_LEN]) {
    // ...
    int required_perm = PERM_READ;
    if (flags & O_WRONLY || flags & O_RDWR) {
        required_perm |= PERM_WRITE;
    }
    if (!fs_check_permission(args[0], required_perm)) {
        printf("Permission denied\n");
        return;
    }
    // ...
}
```

۳.۶.۳ به‌روزرسانی دستور read

```
void cmd_read(int argc, char args[][MAX_COMMAND_LEN]) {
    if (!fs_check_permission(current_filename, PERM_READ)) {
        printf("Permission denied: no read access\n");
        return;
    }
    // ...
}
```

۴.۶.۳ به‌روزرسانی دستور write

```
void cmd_write(int argc, char args[][MAX_COMMAND_LEN]) {
    if (!fs_check_permission(current_filename, PERM_WRITE)) {
        printf("Permission denied: no write access\n");
        return;
    }
    // ...
}
```

۵.۶.۳ به‌روزرسانی دستور rm

```
void cmd_rm(int argc, char args[][MAX_COMMAND_LEN]) {
    if (!fs_check_permission(args[0], PERM_WRITE)) {
        printf("Permission denied: no write access\n");
        return;
    }
    // ...
}
```

۷.۳ تست عملکرد

۱.۷.۳ سناریوی تست

```
\$./myfs
 Loading disk: filesys.db
 Disk loaded (version 1, 1 files)


    Welcome to MyFileSystem

Type 'help' for commands
myfs:root> useradd parsa
 Group 'parsa' created (gid=2)
 User 'parsa' created (uid=2, gid=2)

myfs:root> create secret.txt 600
 File secret.txt created
 File secret.txt created successfully

myfs:root> getfacl secret.txt

 Access Control List for 'secret.txt':

  File:    secret.txt
  Owner:   root (uid=0)
  Group:   root (gid=0)
  Mode:    600

  user::rw-
  group::---
  other::---


myfs:root> su parsa
 Switched to user 'parsa' (uid=2)

myfs:parsa> open secret.txt r
 Permission denied

myfs:parsa> su root
 Switched to user 'root' (uid=0)
```

```
myfs:root> chown parsa secret.txt
 Owner of 'secret.txt' changed to 'parsa'

myfs:root> su parsa
 Switched to user 'parsa' (uid=2)

myfs:parsa> open secret.txt
 File secret.txt opened (fd=0)
 File secret.txt opened (fd=0)

myfs:parsa [secret.txt]> close
 File closed (fd=0)
 File secret.txt closed

myfs:parsa> exit
 Goodbye!

 Filesystem saved and closed
```

۸.۳ لیست دستورات جدید

دستور	توضیحات
useradd <username>	ایجاد کاربر جدید
userdel <username>	حذف کاربر
users	نمایش کاربران
groupadd <groupname>	ایجاد گروه جدید
groupdel <groupname>	حذف گروه
groups	نمایش گروه‌ها
usermod -aG <user> <group>	افزودن کاربر به گروه
chmod <mode> <file>	تغییر سطح دسترسی
chown <user>:<group> <file>	تغییر مالک و گروه
chgrp <group> <file>	تغییر گروه
getfacl <file>	نمایش لیست دسترسی
su <username>	تغییر کاربر
whoami	نمایش کاربر فعلی

جدول ۱: دستورات جدید اضافه شده به سیستم فایل

با پیاده‌سازی مکانیزم‌های امنیتی:

- هر فایل دارای یک مالک (owner) و یک گروه (group) است
- سطح دسترسی به سه بخش owner، group و others تقسیم می‌شود
- هر بخش می‌تواند دسترسی خواندن (r=4)، نوشتن (w=2) و اجرا (x=1) داشته باشد
- کاربر root به تمام فایل‌ها دسترسی کامل دارد

- دستورات مدیریت کاربران و گروه‌ها فقط توسط root قابل اجرا هستند
- قبل از هر عملیات روی فایل، سطح دسترسی کاربر بررسی می‌شود