دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

تمرین سری چهارم - عملی

پارسا ملکیان - ۴۰۲۱۷۱۰۷۵
فاطمه شفیعی - ۴۰۲۱۱۰۸۸۷

سیستم‌های عامل
دکتر اسدی

سوال ۱

در این تمرین نخست ساختار داده‌ها را همانطور که گفته شده بود پیاده سازی کردیم و تعریف ساختار هر ترد و کانتکست آن به شکل زیر در فایل uthread.h قرار گرفتند:

```
struct context {
  uint64 ra;
  uint64 sp;
  uint64 s0; uint64 s1; ... uint64 s11;
};

struct thread {
  char stack[STACK_SIZE];   //
  int state;                //    : FREE, RUNNING, RUNNABLE
  struct context context;   //
{;
```

- `stack`: حافظه‌ی اختصاصی ترد برای نگهداری داده‌ها و فراخوانی‌ها.
- `state`: وضعیت ترد.
- `context`: مقادیر ثبات‌ها که توسط `uthread_switch` استفاده می‌شود.

سپس به پیاده سازی توابع مورد نیاز پرداختیم که توضیح مختصری از عملکرد آنها در زیر آماده است:

thread_init

این تابع اولین ریسه $(\text{thread } 0)$ را که ریسه‌ی main است، در حالت `RUNNING` قرار می‌دهد.

```
void
thread_init(void)
{
  current_thread = &all_thread[0];
  current_thread->state = RUNNING;
}
```

thread_create

با استفاده از این تابع، ریسه‌های جدید ایجاد شدند:

- بررسی می‌کند که ریسه‌ی آزاد موجود است یا خیر.
- کانتکست ریسه را صفر می‌کند و آدرس استک را در `sp` تنظیم می‌کند.
- آدرس تابع ریسه را در `ra` قرار می‌دهد.
- وضعیت ریسه را به `RUNNABLE` تغییر می‌دهد.

```
void
thread_create(void (*func)())
{
  int tid;

  for (tid = 0; tid < MAX_THREAD; tid++) {
    if (all_thread[tid].state == FREE) break;
  }

  if (tid == MAX_THREAD) {
    printf("create_thread: no free thread\n");
    return;
  }

  struct thread *t = &all_thread[tid];

  // allocate space to the context
  int sz = sizeof(struct context);
  memset(&t->context, 0, sz);

  // allocate space to the stack
  uint64 sp = (uint64)(t->stack + STACK_SIZE);
  t->context.sp = sp;

  // runnable
  t->context.ra = (uint64)func;
  t->state = RUNNABLE;
}
```

thread_schedule

وظیفه زمان‌بندی ریسه‌ها بر عهده این تابع است:

- از ریسه‌ی جاری شروع کرده و در بین ریسه‌های ۱ تا ۳ دنبال ریسه‌ی RUNNABLE می‌گردد.
- اگر ریسه‌ی قابل اجرا یافت شد، وضعیت ریسه جاری را به RUNNABLE تغییر می‌دهد و ریسه‌ی بعدی را RUNNING می‌کند.
- سپس uthread_switch برای سوئیچ کانتکست فراخوانی می‌شود.

```
void
thread_schedule(void)
{
  int i, next = -1;
  int indx = current_thread - all_thread;
  // find a runnable thread
  for (i = indx; i < indx + MAX_THREAD ; i++) {
    if ((all_thread[i % (MAX_THREAD - 1) + 1].state == RUNNABLE)) {
      next = i % (MAX_THREAD - 1) + 1;
      break;
    }
```

```
  }

  // no runnable threads, nothing to schedule right now
  if (next == -1) {
      printf("thread_schedule: no runnable threads\n");
      all_thread->state = RUNNING;
      thread_switch(&current_thread->context, &all_thread->context);
      current_thread = all_thread;
      return;
  }

  struct thread *told = current_thread;
  struct thread *tnext =  &all_thread[next];

  told->state = (told->state == RUNNING) ? RUNNABLE : told->state;
  tnext->state = RUNNING;
  current_thread = tnext;
  thread_switch(&told->context, &tnext->context);
```

thread_yield

این تابع باعث می‌شود ریسه‌ی جاری خود را به حالت آماده اجرا درآورد و تابع زمان‌بندی را صدا بزند.

```
void
thread_yield(void)
{
  if (current_thread->state == RUNNING) current_thread->state = RUNNABLE;
  thread_schedule();
}
```

thread_c ، thread_b ، thread_a

- هر ریسه ابتدا با متغیرهای `a_started b_started c_started` شروع خود را علامت‌گذاری می‌کند و تا وقتی همه ریسه‌ها شروع نشده‌اند، صبر می‌کند.
- سپس حلقه‌ای از ۰ تا ۹۹ اجرا می‌شود و در هر مرحله، خروجی شمارنده ریسه چاپ می‌شود.
- بعد از اتمام حلقه، ریسه به حالت `FREE` تغییر می‌کند و پیام اتمام چاپ می‌شود.

```
void
thread_a(void)
{
  a_started = 1;
  printf("thread_a started\n");
  while (!(a_started && b_started && c_started)) {
    thread_yield();
  }

  for (; a_n < 100; a_n++) {
    printf("thread_a %d\n", a_n);
```

```
    // let others run
    thread_yield();
  }
  current_thread->state = FREE;
  printf("thread_a: exit after 100\n");
  thread_schedule();
}
```

در نهایت نتیجه اجرای دستور uthread در سیستم عامل داده شده به شکل زیر شد که مطابق خواسته سوال است:

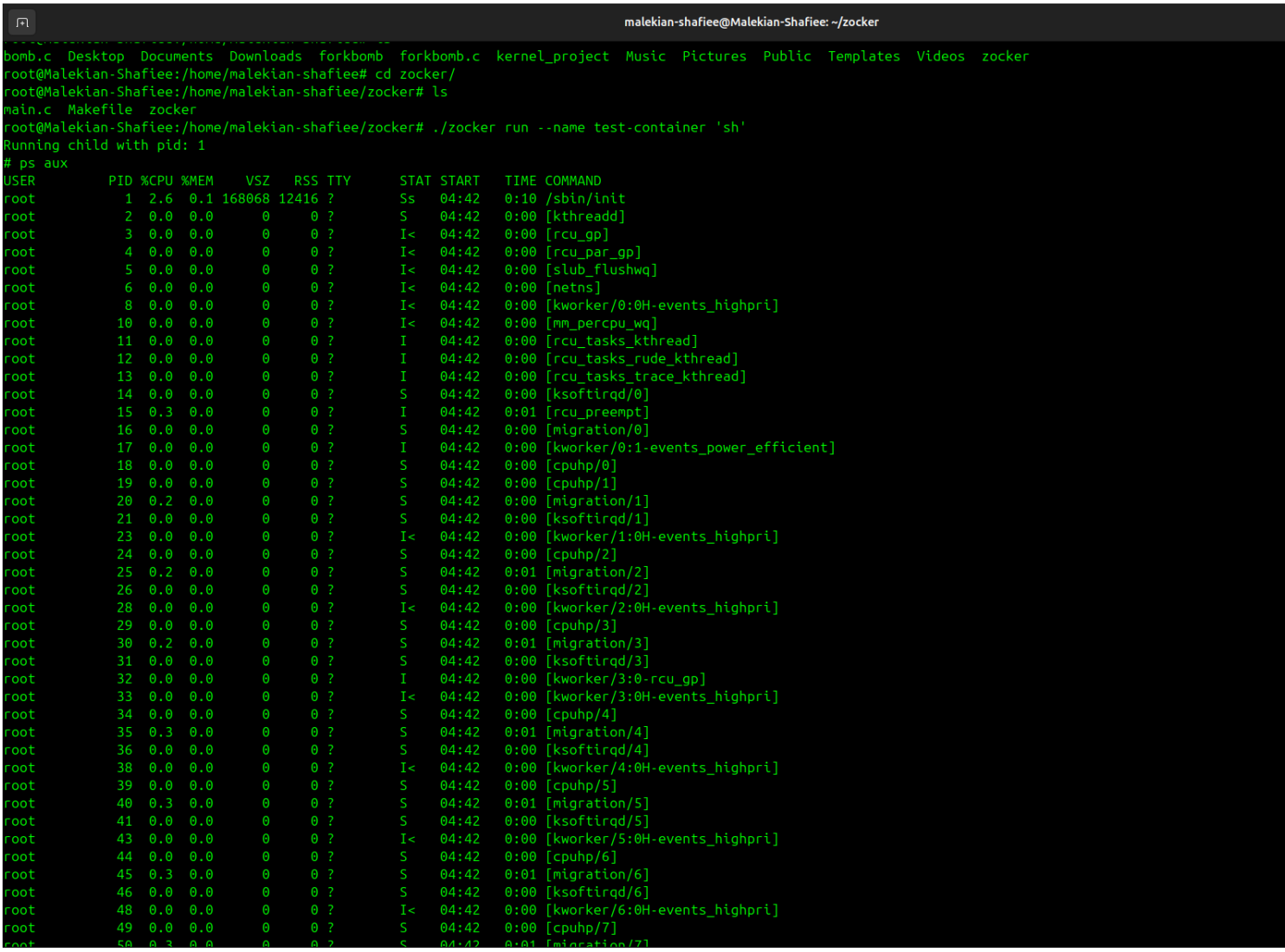
```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
thread_c 2
thread_a 2
thread_b 2
thread_c 3
```

```
thread_c 95
thread_a 95
thread_b 95
thread_c 96
thread_a 96
thread_b 96
thread_c 97
thread_a 97
thread_b 97
thread_c 98
thread_a 98
thread_b 98
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
```

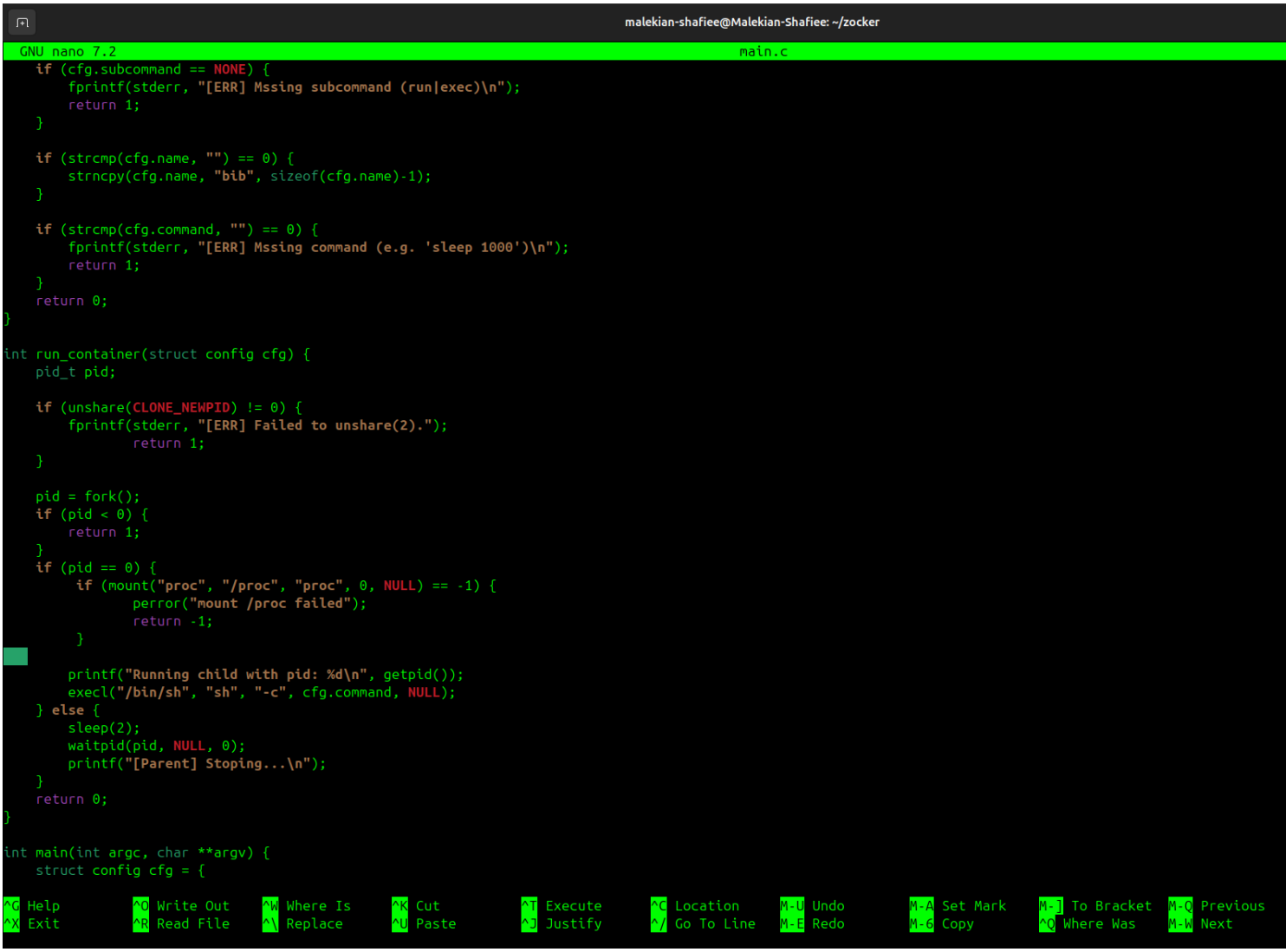
سوال ۲

بخش الف و ب: شروع کار و کشف مشکل اول اول کد رو از git گرفتم و به t۲ tag رفتم. بعد کانتینر رو با دستور `''sh -containertest -name- run ./zocker` اجرا کردم. وقتی داخل کانتینر دستور ps رو زدم، دیدم که تمام پردازه‌های سیستم host رو نشان میده. مشکل: متوجه شدم که دستور ps اطلاعاتش رو از /proc می‌خونه و چون /proc از سیستم اصلی mount شده بود، همه پردازه‌ها رو نشون میداد.

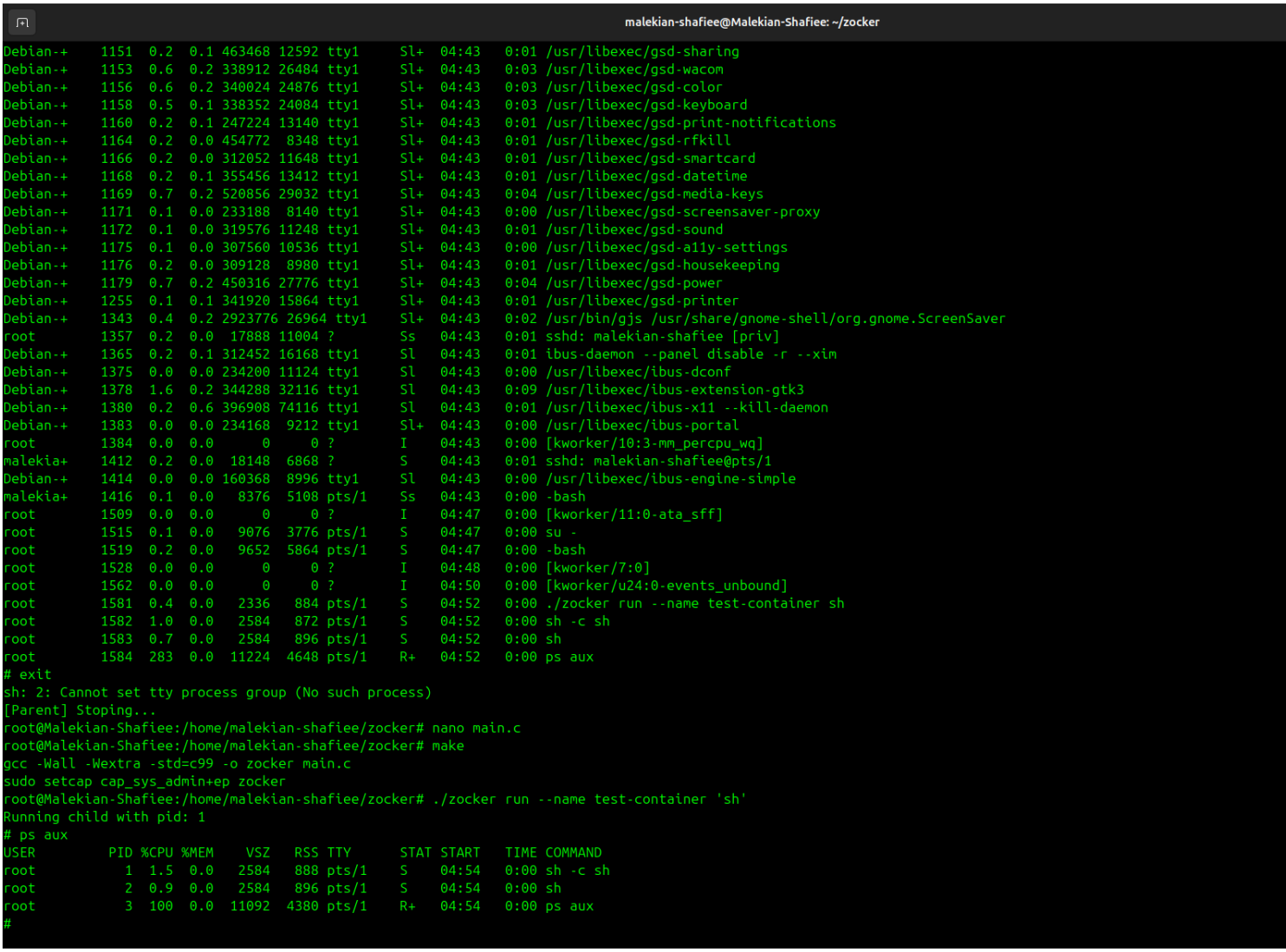


بخش ج: حل مشکل ps برای حل این مشکل، باید یک /proc جدید برای namespace خودمون mount می‌کردیم. کدی که نوشتم:

```c
if (mount("proc", "/proc", "proc", 0, NULL) == -1) {
    perror("mount /proc failed");
    exit(1);
}
```



حالا ps فقط پردازه‌های داخل کانتینر رو نشون میداد



بخش د: مشکل جدید با df وقتی mount رو درست کردم، دیدم که دستورات دیگه مثل df خطا میدن. حتی بعد از خروج از کانتینر، سیستم host مشکل پیدا کرده بود

فهمیدم که به خاطر mount propagation بوده. به طور پیش‌فرض، تغییرات mount به parent namespace هم منتشر میشه .(MS_SHARED) وقتی ما /proc رو unmount کردیم، روی host هم اثر گذاشت.

بخش ه: جلوگیری از انتشار mount برای حل این مشکل، باید mount propagation رو به private تبدیل می‌کردم: همچنین باید هم NEWPID و هم NEWNS رو unshare میکردیم

```
1
2 if (unshare(CLONE_NEWPID | CLONE_NEWNS) != 0) {
3     fprintf(stderr, "[ERR] Failed to unshare(2).");
4     return 1;
5 }
6
7 if (mount(NULL, "/", NULL, MS_PRIVATE | MS_REC, NULL) == -1) {
8     perror("make-rprivate failed");
9     return -1;
```

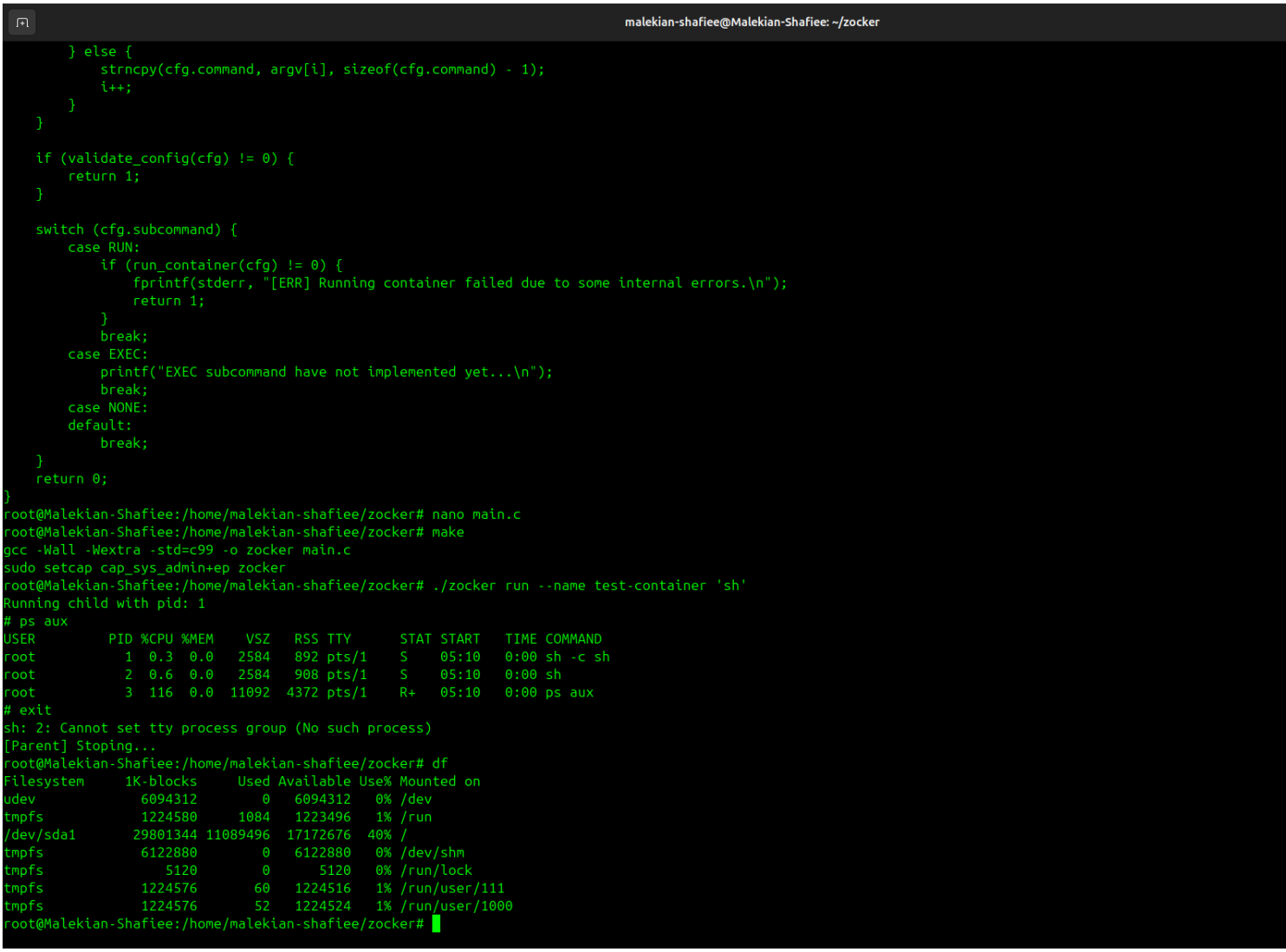
```
}
```

بخش دوم بلافاصله بعد از آنشیر نوشته شده

پرچم‌ها:

- `MS_REC`: روی همه زیرشاخه‌ها اعمال بشه
- `MS_PRIVATE`: تغییرات فقط تو namespace خودمون بمونه

حالا سیستم host دیگه تحت تاثیر قرار نمی‌گرفت.



بخش و: اضافه کردن Namespace UTS برای اینکه hostname هر کانتینر جدا باشه، `CLONE_NEWUTS` رو به unshare اضافه کردم:

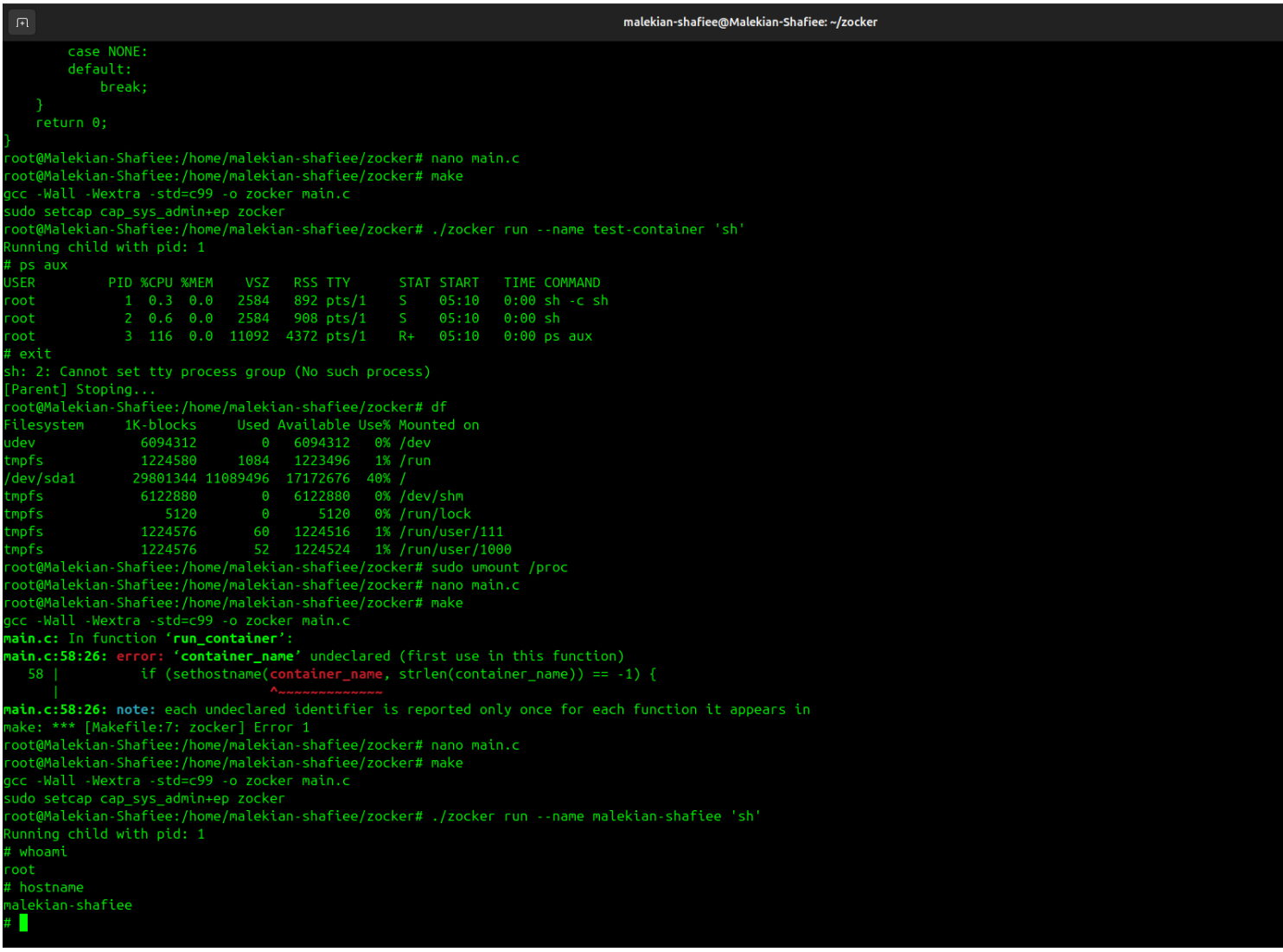
```
if (unshare(CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUTS) == -1) {
    perror("unshare failed");
    return -1;
```

```
}
```

بخش ز: تنظیم hostname با استفاده از sethostname اسم کانتینر رو به عنوان hostname تنظیم کردم:

```
if (sethostname(container_name, strlen(container_name)) == -1) {
    perror("sethostname failed");
    exit(1);
}
```

وقتی داخل کانتینر hostname رو می‌زدم، اسم کانتینر رو نشون میداد نه hostname سیستم اصلی.



بخش ح: User و Namespace Time در آخر namespace های کاربر و زمان رو هم اضافه کردم:

```
if (unshare(CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUTS | CLONE_NEWUSER) == -1) {
    perror("unshare failed");
    return -1;
```

```
4 }
```

کد کامل بعد از تغییرات داخل فایل های ارسالی در پوشه q۲ قرار دارد.

سوال ۳

در این تمرین یک سیستم مدیریت فایل ساده پیاده سازی کردم که روی یک فایل معمولی کار می‌کند و اندازه ثابت دارد.

ساختار کلی

سیستم از یک فایل به نام filesys.db استفاده می‌کند که اندازه آن ثابت است و در واقع نقش یک دیسک مجازی را دارد. این فایل به بلاک‌های ۵۱۲ بایتی تقسیم شده است.

در ابتدای فایل یک SuperBlock قرار دارد که حاوی اطلاعات کلی سیستم است شامل number magic برای اعتبارسنجی، نسخه، تعداد فایل‌ها، تعداد بلاک‌های کل و استفاده شده. بعد از آن یک آرایه ثابت از FileEntry ها وجود دارد که هرکدام اطلاعات یک فایل را نگه می‌دارند.

هر FileEntry شامل نام فایل، سایز، بلاک شروع، تعداد بلاک‌ها، دسترسی‌ها و زمان ساخت و ویرایش است. همچنین یک فیلد is_used دارد که مشخص می‌کند این ورودی استفاده شده یا خالی است.

مدیریت فضای خالی

برای مدیریت فضای خالی از یک لیست پیوندی استفاده کردم که هر نود آن نشان دهنده یک بازه پیوسته از بلاک‌های خالی است. وقتی فایل جدیدی ساخته می‌شود از ابتدای این لیست بلاک‌های لازم اختصاص داده می‌شود و وقتی فایلی پاک می‌شود بلاک‌های آن به لیست اضافه می‌شوند.

تابع allocate_blocks روی لیست پیوندی حرکت می‌کند تا اولین بلاک خالی با فضای کافی را پیدا کند. اگر پیدا کرد آن قسمت را از لیست حذف یا کوچک می‌کند و آدرس بلاک شروع را برمی‌گرداند.

تابع free_blocks یک نود جدید با آدرس و تعداد بلاک‌های آزاد شده می‌سازد و آن را به ابتدای لیست اضافه می‌کند.

پایداری داده‌ها

برای اینکه داده‌ها بعد از بستن برنامه حفظ شوند دو تابع save_metadata و load_metadata نوشتم. تابع save در ابتدای فایل دیسک SuperBlock و سپس کل آرایه file_table را می‌نویسد.

تابع load در شروع برنامه این اطلاعات را از دیسک می‌خواند و سپس لیست فضای خالی را بازسازی می‌کند. برای بازسازی ابتدا فرض می‌کند همه فضا خالی است و سپس با چک کردن فایل‌های موجود بلاک‌های استفاده شده را از لیست خالی حذف می‌کند.

عملیات فایل

تابع fs_create یک ورودی خالی در file_table پیدا می‌کند و اطلاعات فایل جدید را در آن می‌نویسد. در این مرحله فقط metadata ساخته می‌شود و هنوز بلاکی اختصاص داده نشده است.

تابع fs_open نام فایل را جستجو می‌کند و در صورت پیدا شدن یک descriptor file برمی‌گرداند. از یک آرایه open_files برای نگهداری ارتباط بین fd و index فایل در file_table استفاده کردم.

تابع fs_write ابتدا محاسبه می‌کند چند بلاک برای نوشتن نیاز است. اگر فایل بلاک نداشت از allocate_blocks بلاک‌های لازم را می‌گیرد و اگر بلاک‌های فعلی کافی نبود بلاک اضافی درخواست می‌کند. سپس با fseek به موقعیت مناسب در فایل دیسک می‌رود و داده را می‌نویسد.

تابع fs_read مشابه write عمل می‌کند با این تفاوت که فقط می‌خواند و اگر درخواست خواندن بیشتر از سایز فایل بود آن را محدود می‌کند.

تابع fs_shrink فایل را به سایز کوچکتری تبدیل می‌کند. بلاک‌های اضافی را محاسبه کرده و با free_blocks آنها را آزاد می‌کند و سایز فایل را به روز می‌کند.

تابع fs_delete پس از پیدا کردن فایل بلاک‌های آن را آزاد می‌کند و فیلد is_used را صفر می‌کند. همچنین descriptor file های مربوط به آن فایل را می‌بندد.

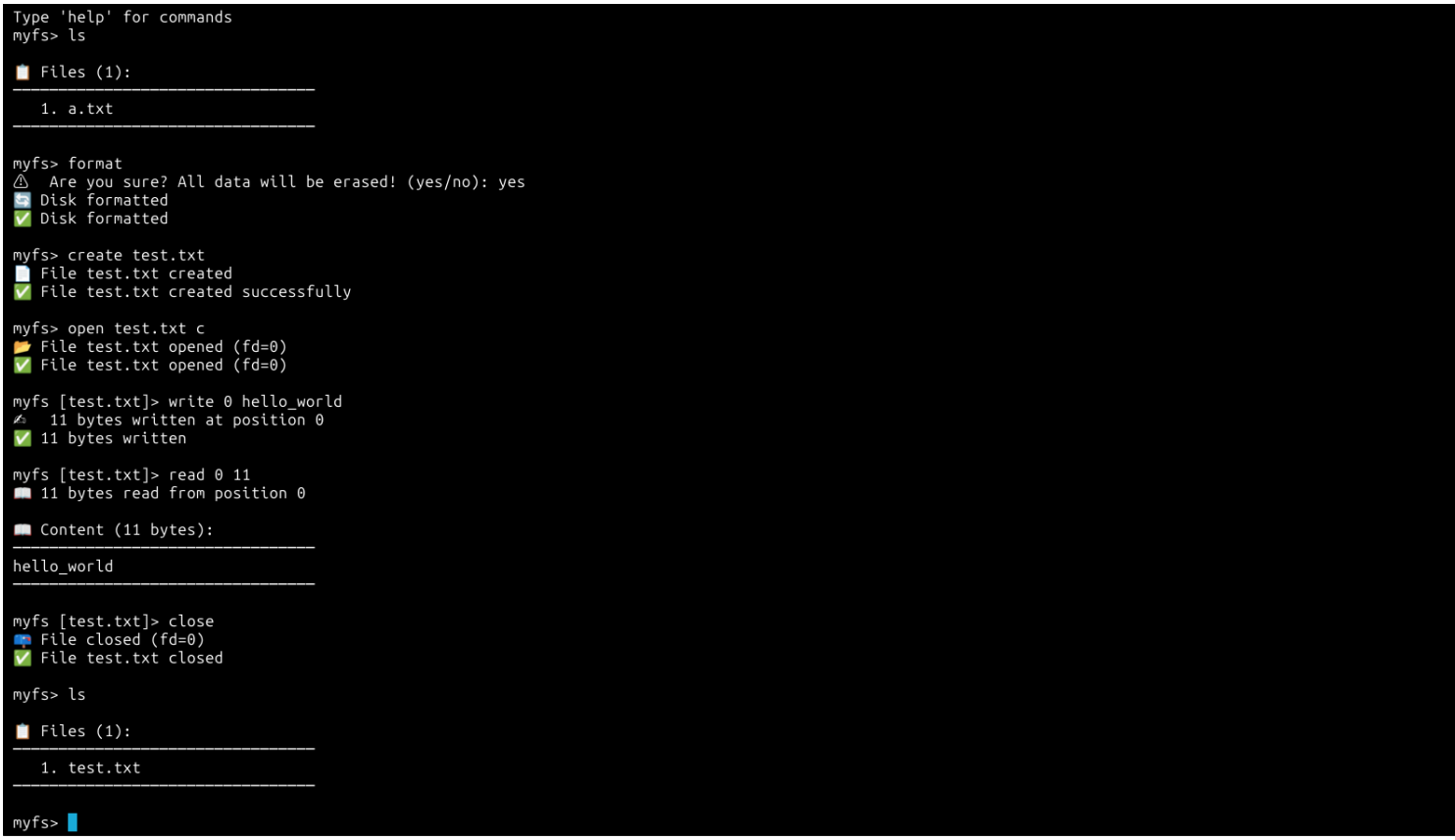
رابط کاربری

یک CLI ساده پیاده سازی کردم که در یک حلقه دستورات کاربر را می‌خواند و اجرا می‌کند. دستوراتی مثل create برای ساخت فایل، open برای بازکردن، read و write برای خواندن و نوشتن، ls برای لیست فایل‌ها و stat برای نمایش وضعیت فایل سیستم.

برای پردازش دستورات از strtok استفاده کردم که رشته ورودی را به کلمات جداگانه تقسیم می‌کند. اولین کلمه نام دستور و بقیه آرگومان‌ها هستند.

در prompt نشان داده می‌شود که آیا فایلی باز است یا نه و اگر باز باشد نام آن نمایش داده می‌شود.

نمونه استفاده:



کد کامل، همراه با فایل readme و دستور help برای استفاده راحت از cli کد، در پوشه q۳ قرار دارد.