دانشگاه صنعتی اصفهان

دانشکده مهندسی برق و کامپیوتر

پروژه دوم درس سیستم‌های عامل ۱

نیمسال تحصیلی پاییز ۱۴۰۰

دکتر محمّدرضا حیدرپور - دکتر زینب زالی

دستیاران آموزشی:

محمّد روغنی - مجید فرهادی - عرفان مظاهری - دانیال مهرآیین - محمّد نعیمی

در این پروژه شما با نحوه پیاده‌سازی یک Loadalbe Kernel Module آشنا می‌شوید. شما باید یک کاراکتر دیوایس برای سیستم عامل لینوکس پیاده‌سازی کنید که یک بانک کوچک را شبیه‌سازی میکند.

۱ LKM چیست؟

یکی از قابلیت‌های خوب لینوکس امکان توسعه کرنل هنگام بالا بودن سیستم عامل است. یعنی میتوان حین اجرای سیستم عامل قابلیت‌هایی را به آن اضافه یا از آن کم کرد (می‌توانید در مورد Uptime بالای سرورهای لینوکسی تحقیق کنید). به قطعه کدهایی که حین اجرا به کرنل افزوده میشوند ماژول گفته می‌شود. کرنل لینوکس از انواع مختلف ماژول (مثلاً درایورها) پشتیبانی میکند. هر ماژولی از Object Code تشکیل شده‌است که می‌تواند به صورت پویا به کرنل لینک شود، بدون نیاز به کامپایل دوباره کل کرنل. پس از اضافه شدن یک ماژول به کرنل، اپلیکیشن‌های فضای کاربر می‌توانند از آن ماژول استفاده کنند.

چرا LKM را انتخاب کردیم؟

شما باید تجربه کار روی لینوکس به عنوان یک سیستم عامل متن‌باز و کرنل عظیم آن‌را داشته باشید و یکی از راحت‌ترین نقاط ورود به این کرنل بزرگ، LKM است.

دیوایس‌ها و درایورها

تقریباً هر عملیات سیستمی نهایتاً با یک دستگاه فیزیکی کار خواهد داشت. تمامی عملیات‌های کنترل دستگاه (به جز دستگاه‌هایی مثل پردازنده و حافظهٔ اصلی) توسط قطعه کدهایی انجام میشود که مخصوص به دستگاه هدف (دستگاهی که عملیات رو آن انجام می‌شود) است. به این قطعه کدها درایور گفته میشود. کرنل باید برای تمامی دستگاه‌های متصل به سیستم درایور مخصوص به خودشان را داشته باشد (مثلاً برای ماوس و کیبورد و درایو باید درایور داشته باشد).

انواع دیوایس

در لینوکس به طور کلی سه مدل دستگاه تعریف میشود. هر ماژول هم معمولاً فقط تحت یکی از این سه مدل توسعه می‌یابد که نتیجهٔ آن سه دسته ماژول کاراکتری (Char module) ، بلوکی (Block module) و شبکه‌ای (Network module) است. البته میتوانیم این دسته‌بندی را رعایت نکنیم و ماژولی بنویسیم که بتواند قابلیت‌هایی از هر سه دسته داشته باشد اما این ماژول مقیاس پذیر و توسعه پذیر نخواهد بود.

کاراکتر دیوایس، نقطهٔ تمرکز پروژه

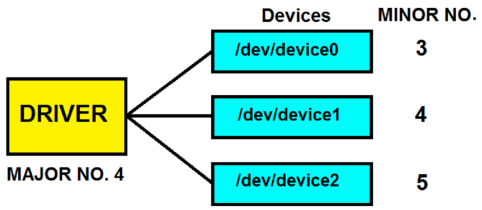
یک دستگاه کاراکتری دستگاهی‌ست که بتوان با آن مثل یک فایل رفتار کرد؛ یعنی مثل یک جریانی از بایت‌ها (Stream of Bytes). یک درایور کاراکتری چنین رفتار فایل مانندی را برای این دستگاه کنترل و پیاده‌سازی می‌کند. این درایورهای کاراکتری معمولاً فراخوانی‌های سیستمی بازکردن (open)، بستن (close)، خواندن (read) و نوشتن (write) را پیاده‌سازی می‌کنند. به عنوان مثال کنسول متنی (/dev/console) و پورت‌های سریال (/dev/ttyS0) و مشابه‌های آن) دستگاه‌های کاراکتری هستند.

دسترسی به دستگاه‌های کاراکتری به کمک گره‌های فایل‌سیستم (Filesystem Nodes) انجام می‌شود. تنها تفاوت قابل توجه بین دستگاه‌های کاراکتری و فایل‌های معمولی این است که در فایل‌های معمولی می‌توان به عقب و جلو حرکت کرد اما معمولاً دستگاه‌های کاراکتری کانال‌های داده‌ای هستند که فقط به صورت سری (Sequentially) قابل دسترسی هستند. البته دستگاه‌های کاراکتری‌ای هم وجود دارد که مثل نواحی داده‌ای (Data area) رفتار میکنند و میتوان در آنها به عقب و جلو حرکت کرد.

اعداد ماژور و مینور

به هر درایور در سیستم یک عدد یکتا تخصیص داده می‌شود که به آن Major Number می‌گویند. بدین ترتیب موقع load هر درایور در سیستم باید یک عدد ماژور آزاد به آن تخصیص داده شود. توجه داشته باشید که دیوایس‌های سخت‌افزاری مختلفی می‌توانند از طریق یک نوع درایور کنترل شوند. مثلاً تصور کنید دو هارد اکسترنال مشابه به سیستم شما وصل باشد. این دو هارد اکسترنال از یک درایور یکسان استفاده می‌کنند اما سیستم باید راهی جهت تفکیک این دو دیوایس داشته باشد. به همین دلیل عدد دیگری با نام Minor Number برای دیوایس‌های مختلفی که از یک درایور واحد استفاده می‌کنند در نظر گرفته می‌شود.

کرنل از عدد ماژور استفاده میکند تا درایور مرتبط را پیدا کند و درایور از عدد مینور جهت کار با دستگاه مشخصی استفاده میکند.



یک فایل خاص! (و روش ساختن آن)

یکی از مفاهیم مهم در سیستم‌عامل‌های مبتنی بر یونیکس مفهوم فایل بودنِ تقریباً همه چیز است. یعنی منابع ورودی/خروجی مختلفی (مثل اسناد، دایرکتوری‌ها، درایوها، مودم، کیبورد، پرینتر و حتی برخی ipc ها و ارتباطات شبکه‌ای) وجود دارد که همگی جریان‌های بایتی ساده‌ای هستند. مزیت چنین رویکردی این است که ابزارها و API های یکسانی را میتوان برای دسترسی و ارتباط با چندین منبع مختلف استفاده کرد.

البته میتوان این مفهوم را دقیقتر هم بیان کرد و گفت هر چیزی یک File Descriptor است. چرا که هنگام باز کردن یک فایل معمولی یا ایجاد پایپ‌های ناشناس یا ساخت سوکت شبکه، File Descriptor هایی ساخته می‌شود که راه ارتباطی و رابط بین کد با آن منبع خواهد بود.

دستگاه‌های کاراکتری از طریق اسم‌شان در فایل‌سیستم قابل دسترسی هستند و میتوان با آنها مثل یک فایل رفتار کرد. این اسامی را "فایل‌های خاص"، "فایل‌های دستگاهی" یا حتی "گره‌هایی در درخت فایل‌سیستم" گوییم. اما این فایل خاص کجاست؟ معمولاً فایل درایور مرتبط با هر دیوایس در دایرکتوری "/dev" قرار دارد. `ls /dev -l` را اجرا کنید تا فایل‌های مربوط به ماژول‌های کنونی سیستم‌تان را مشاهده کنید. همانطور که می‌بینید اولین کاراکتر از رشته permission هر فایل، مشخص‌کننده نوع دیوایس یا فایل ماژول است (c به معنی دیوایس کاراکتری و b به معنی دیوایس بلوکی). همچنین غیر از نام فایل، دو ستون عددی وجود دارد که یکی بیانگر عدد ماژور و دیگری بیانگر عدد مینور دیوایس است. همانطور که گفتیم دیوایس‌های مختلفی ممکن است از یک ماژول استفاده کنند که بدین ترتیب همه دارای یک عدد ماژور ولی عددهای متفاوت مینور هستند.

هرگاه دیوایسی به سیستم اضافه می‌شود باید حتما فایل درایور متناظرش در شاخه /dev قرارگیرد و در واقع از طریق نام همین فایل است که در کد اپلیکیشن می‌توانیم مشخص کنیم با کدام دیوایس کار داریم و عملیات open، close، read و write را روی چه دیوایسی انجام می‌دهیم. البته این فایل برای پروژهٔ ما باید توسط خودمان ساخته شود. ساخت این فایل به کمک دستور "mknod" انجام می‌شود. می‌توانید به کمک man page مرتبط به این دستور اطلاعات خوبی در مورد نحوهٔ کار با آن به دست آورید. در عین حال روش‌هایی جهت ساخت این فایل با استفاده از کدنویسی هم وجود دارد.

توابع استاندارد (libc)

درایورها در فضای کرنل اجرا میشوند. به همین علت نمی‌توان از برخی توابع معروف برای نوشتن کد درایورها استفاده کرد. مثلاً به جای printf از printk استفاده می‌شود که در لاگ کرنل رشتهٔ مورد نظر را چاپ می‌کند (به جای خروجی استاندارد). در کد نمونه، با این تابع آشنا خواهید شد.

ذات ری‌اکتیو بودن درایورها

برنامه‌های معمولی از بالا (از اولین خط) شروع به اجرا شده و در پایین (در آخرین خط) خاتمه می‌یابند. درایورها چنین روش اجرایی ندارند. درایورها به رخدادهایی مرتبط با دیوایس متناظرشان پاسخگو هستند. رخدادهایی که در این پروژه مورد استفاده قرار میگیرند رخدادهای مربوط به load و unload ماژول، باز و بسته کردن فایل درایور، خواندن از و نوشتن به فایل درایور است.

۲ نوشتن یک دیوایس ساده

در ادامه یک دیوایس ساده که یک پیغام را به کاربر نشان میدهد مینویسیم. توجه کنید که پروژهٔ اصلی تفاوت زیادی با این دیوایس نمونه نخواهد داشت.

لینوکس هدرز و Makefile برای ماژول

برای ساختن ماژول به هدرفایل‌های مخصوصی نیاز داریم. نام این هدرفایل‌ها "Linux Headers" است. این هدرفایل‌ها به کامپایلر کمک می‌کنند تا بررسی کند که آیا توابع مرتبط با کرنل (در این پروژه: جهت تولید ماژول) به درستی استفاده شده‌اند. همین هدرفایل‌ها هستند که ارتباط بین اجزای کرنل را میسر می‌سازند و همچنین آنها بین فضای کاربر و فضای کرنل به عنوان یک اینترفیس رفتار می‌کنند. شما باید برای نسخهٔ لینوکس خودتان این هدرفایل‌ها را دریافت و نصب کنید. مثلاً برای اوبونتو می‌توان از دستور زیر استفاده کرد.

```
sudo apt install linux-headers-$(uname -r)
```

پس از نصب لینوکس هدرز باید در دایرکتوری پروژه، یک Makefile ایجاد کرد. محتویات این فایل به فرمت زیر است. (توجه کنید که در این مثال نام فایلی که در آن کد ماژول را مینویسیم mycode.c است)

```
obj-m = mycode.o
all:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

در اینجا فلگ `-C` جهت انتقال به دایرکتوری مشخص شده اضافه شده است و "M" هم جهت بازگشت به دایرکتوری پروژه هنگام تولید فایل‌های نهایی است. کلمه‌های "modules" و "clean" هم مربوط به target این Makefile هستند.

نیازی به یادگیری نحوهٔ نوشتن Makefile و علی‌الخصوص Makefile جهت ساختن LKM ندارید. هرچند که جهت مطالعهٔ آزاد می‌تواند مبحث جالبی باشد.

بعد از ایجاد Makefile ، می‌توانید به کمک دستورهای "make" و "make clean" پروژه را بسازید یا فایل‌های ساخته شده را پاک کنید.

File Operations، Load و Unload

همانطور که گفته شد درایورها ذات ری‌اکتیو دارند و به رخدادها پاسخ می‌دهند. توابعی که برای پاسخ به رویدادها به کرنل معرفی می‌کنیم قالب مشخصی دارند (در کد نمونه با آنها آشنا می‌شوید). جهت معرفی توابع مرتبط با load و unload ماژول از ماکروهای module_init و module_exit استفاده می‌کنیم و جهت معرفی توابع باز و بسته کردن دستگاه (مثلاً open کردن فایل خاصی که می‌سازیم) و خواندن از و نوشتن به دستگاه از ساختمان دادهٔ file_operations استفاده می‌کنیم.

کد نمونه دستگاه ساده:

```c
#include <linux/init.h> // For module init and exit
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h> // For fops
#include <linux/uaccess.h>
//#include <string.h> // Can't use it!

#define DEVICE_NAME "iut_device"
MODULE_LICENSE("GPL");

// FILE OPERATIONS
static int iut_open(struct inode*, struct file*);
static int iut_release(struct inode*, struct file*);
static ssize_t iut_read(struct file*, char*, size_t, loff_t*);

static struct file_operations fops = {
   .open = iut_open,
   .read = iut_read,
   .release = iut_release,
};

// Why "static"? --> To bound it to the current file.
static int major; // Device major number. Driver reacts to this major number.

// Event --> LOAD
static int __init iut_init(void) {
    major = register_chrdev(0, DEVICE_NAME, &fops); // 0: dynamically assign a major number ||| name is
displayed in /proc/devices ||| fops.
    if (major < 0) {
        printk(KERN_ALERT "iut_device load failed.\n");
        return major;
    }
    printk(KERN_INFO "iut_device module loaded: %d\n", major);
    return 0;
}

// Event --> UNLOAD
static void __exit iut_exit(void) {
    unregister_chrdev(major, DEVICE_NAME);
    printk(KERN_INFO "iut_device module unloaded.\n");
}

// Event --> OPEN
static int iut_open(struct inode *inodep, struct file *filep) {
   printk(KERN_INFO "iut_device opened.\n");
   return 0;
}

// Event --> CLOSE
static int iut_release(struct inode *inodep, struct file *filep) {
   printk(KERN_INFO "iut_device closed.\n");
   return 0;
}

// Event --> READ
static ssize_t iut_read(struct file *filep, char *buffer, size_t len, loff_t *offset) {
    char *message = "IUT OS project 2!";
    int errors = 0;
    errors = copy_to_user(buffer, message, strlen(message));
    return errors == 0 ? strlen(message) : -EFAULT;
}

// Registering load and unload functions.
module_init(iut_init);
module_exit(iut_exit);
```

بررسی مختصر کد نمونه

در کد یک ماژول کاراکتری، توابع پیش‌فرضی وجود دارند که شما باید به صورت اختصاصی با توجه به هدف ماژول آن‌ها را پیاده‌سازی کنید. تابع init هنگام load یک ماژول در سیستم فراخوانی می‌شود لذا در این تابع، عملیات مربوط به رجیستر کردن ماژول در سیستم انجام می‌شود. در مقابل وقتی ماژولی unload می‌شود، تابع exit فراخوانی می‌شود؛ پس در این تابع، مناسب است که ماژول را unregister کرده و عدد ماژور آن را آزاد کنیم. توجه کنید که دو تابع نامبرده به کمک ماکروهای module_init و module_exit در انتهای فایل به کرنل معرفی شده‌اند.

یک ساختارداده بسیار مهم از نوع file_operations در هر ماژول وجود دارد. در این ساختارداده، توابعی که برای ماژول موردنظر در سطح کاربر قابل استفاده است معرفی می‌شود. در واقع API همه ماژول‌های کاراکتری ثابت است اما پیاده‌سازی این API در دست برنامه‌نویس ماژول است. همان‌طور که بیان شد عملیات روی ماژول کاراکتری کاملا شبیه عملیات روی فایل است که این توابع شامل open، close، read و write است. از طریق متغیر file_operations توابع پیاده‌سازی شده توسط برنامه‌نویس ماژول را برای هرکدام از توابع نامبرده معرفی می‌کنیم. مثلا در کد نمونه می‌بینید که تابع open، read و write پیاده‌سازی و نام آن‌ها در file_operations مشخص شده‌است. دقت کنید که متناسب با انتظاری که از ماژول داریم توابع open، read و write و... را پیاده‌سازی می‌کنیم.

شما یک بار ماژول را در سیستم load می‌کنید و اپلیکیشن‌های مختلف و متعدد چندین‌بار (حتی به صورت همزمان) از ماژول load شده استفاده می‌کنند یعنی توابع file_operations را برای آن فراخوانی می‌کنند. پس به ازای هربار open کردن ماژول در یک اپلیکیشن یک File Descriptor برای استفاده از آن ساخته می‌شود و اپلیکیشن پس از آن با استفاده از آن File Descriptor می‌تواند از ماژول بخواند یا به آن بنویسد. مثلا تصور کنید قراراست بافر یک دیوایس سخت‌افزاری از اطلاعاتی که یک اپلیکیشن برای آن ارسال می‌کند پر شود. (write در ماژول) یا اپلیکیشن اطلاعاتی را از بافر سخت‌افزار بخواند (read از ماژول) اینجا چون اطلاعات (داده) بین فضای کاربر و کرنل جابه‌جا می‌شود باید از تواع مخصوص مثل copy_to_user استفاده شود. همچنین نحوه مدیریت داده در ماژول به عهده برنامه‌نویس ماژول است. در کد نمونه، داده پس از دریافت شدن از اپلیکیشن توسط ماژول، به سخت‌افزار ارسال نشده چون این کد مربوط به ماژولی است که به منظور ارتباط با سخت‌افزار نوشته نشده‌است.

درمورد توابع مختلفی که در کد می‌بینید از طریق اینترنت و manual لینوکس می‌توانید اطلاعات خوبی کسب کنید.

استفاده از ماژول ساخته شده

پس از کامپایل کردن ماژول، یک فایل با پسوند "ko" ساخته می‌شود (ko = Kernel Object) . این همان فایلی است که به کرنل متصل خواهد شد و قابلیت‌هایی که نیاز داریم (و کد آنها را نوشته‌ایم) را به کرنل اضافه

خواهد کرد. برای load کردن ماژول (اتصال به کرنل) از دستور "insmod" استفاده می‌شود. اگر این دستور بدون خطا اجرا شود می‌توان ماژول اضافه شده را به کمک دستور "lsmod" مشاهده کرد. برای حذف ماژول از لیست ماژول‌های فعال (unload کردن) نیز از دستور "rmmod" استفاده می‌شود.

تست ماژول

در اینجا نیاز است جهت استفاده از ماژول، فایل دیوایس آن را در /dev ایجاد کنید. این کار از طریق mknod قابل انجام است (عدد ماژور ماژول load شده را می‌توان از طریق جستجوی نام ماژول در فایل /proc/devices به دست آورد). با اینکه فایل ساخته شده به کمک mknod یک فایل خاص است اما ارتباط با آن کار سختی نیست.

به کمک هر زبانی می‌توانیم ماژول نوشته شده را تست کنیم. البته فراموش نکنید که هنگام اجرای برنامهٔ تست باید از sudo استفاده کنیم تا برنامه بتواند فایل درایور را باز کند.

یک نمونه کد با زبان پایتون:

```python
import os
path = "/dev/mynode"
fd = os.open(path, os.O_RDONLY)
data = os.read(fd, 128)
print(f'Number of bytes read: {len(data)}')
print(data.decode())
os.close(fd)
```

خروجی کد:

```
yoyo ~ osprj2 4 test sudo python3 pytest.py
Number of bytes read: 17
IUT OS project 2!
```

دقت کنید خروجی توابع printk در کرنل لاگ قرار می‌گیرد که از طریق مشاهده فایل‌های /var/log یا با استفاده از دستور dmesg قابل مشاهده هستند.

۳ صورت پروژه

دستگاهی طراحی کنید که یک بانک کوچک را شبیه سازی کند. ۱۰۰ نفر در این بانک حساب دارند که با اعداد ۰ تا ۹۹ مشخص می‌شوند. بالانس همهٔ حساب‌ها در ابتدا ۲۰۰۰۰۰۰ واحد پول است.
عملیات روی حساب‌ها از طریق نوشتن به فایل درایور انجام می‌شود و وضعیت حساب‌ها به کمک خواندن از فایل درایور بررسی می‌شود.
فرمت نوشتن به فایل درایور به صورت زیر است:

```
"[Type of transaction],[From],[To],[Amount]" | "r"

[Type of transaction] : “e” | “v” | “b” (Enteghaal, Variz, Bardasht)
[From] : An account number | “-” (0 to 99, Indicating a Variz transaction)
[To] : An account number | “-” (0 to 99, Indicating a Bardasht transaction)
[Amount] : A positive integer indicating the amount of money processed during the transaction.
"r" : Resets all balances to 2000000 units.
```

مثلاً برای انتقال ۱۲ واحد پول از حساب ۱۷ به حساب ۲۲ باید رشتهٔ زیر را در فایل درایور نوشت:

`e,17,22,12`

یا مثلاً برای واریز ۳ واحد پول به حساب ۴۷ باید رشتهٔ زیر را در فایل درایور نوشت:

`v,-,47,3`

یا مثلاً برای برداشت ۴۴ واحد پول از حساب ۹۷ باید رشتهٔ زیر را در فایل درایور نوشت:

`b,97,-,44`

فرمت رشتهٔ خوانده شده از فایل درایور به صورت زیر است:

```
"[Balance of 0],[Balance of 1],...,[Balance of 98],[Balance of 99],"
```

مثلاً در ابتدا نتیجهٔ خواندن از دستگاه باید رشته‌ای شامل صد مرتبه تکرار از زیررشتهٔ زیر باشد:

`2000000,`

هنگام نوشتن ماژول به نکات زیر توجه کنید:

- در صورت عدم رعایت فرمت نوشتن توسط کاربر باید به کمک printk پیغام مناسبی چاپ نمایید.
- در صورت شکست تراکنش (مثلاً کمبود موجودی) باید به کمک printk پیغام مناسبی چاپ نمایید.
- این دستگاه باید بتواند با چندین اپلیکیشن سطح یوزر در ارتباط باشد و بدون اشکال چندین تراکنش همزمان را مدیریت کند.

دستگاه طراحی شده را به کمک یک اپلیکیشن (که خودتان آن را کدنویسی می‌کنید) و طبق سناریوی زیر تست کنید.

در این سناریو حساب شمارهٔ ۰ یک حساب خیریه است. دو درگاه اینترنتی وجود دارد که به این بانک متصل هستند. این درگاه‌ها به صورت همزمان تراکنش‌هایی را برای بانک ارسال میکنند.
تست شما باید در هر مرحله یکی از دیگر حسابها را انتخاب کند (حسابی غیر از حساب شمارهٔ ۰ و به صورت تصادفی) و یک واحد پول از آن حساب تصادفی به حساب شمارهٔ ۰ انتقال دهد. هر درگاه باید یک میلیون بار این تراکنش را انجام دهد.
نهایتاً وضعیت حسابها باید به صورتی باشد که حساب خیریه ۴۰۰۰۰۰۰ واحد پول داشته باشد و دیگر حسابها بالانسی کمتر مساوی با ۲۰۰۰۰۰۰. ضمناً مجموع بالانس تمامی حسابها باید برابر با ۲۰۰۰۰۰۰۰ باشد.

نهایتاً یک شل اسکریپت بنویسید که:

۱. ماژول طراحی شده را کامپایل کند.

۲. ماژول کامپایل شده را Load کند.

۳. به کمک mknod یک فایل درایور به نام iutnode بسازد.

۴. اپلیکیشن تست شما را اجرا کند.

۵. فایل درایور را حذف کند.

۶. ماژول را Unload کند.

۷. فایل‌های خروجی کامپایل را پاکسازی کند.

شرایط زیر را هنگام تحویل رعایت کنید:

۱. فایل‌های زیر از شما دریافت میشود:
یک فایل c که همان فایل ماژول است، یک فایل تست پروژه (اپلیکیشن سطح یوزر) که میتواند به زبان C یا Python باشد و یک فایل شل‌اسکریپت. نام ماژول باید mymodule.c، نام اپلیکیشن تست باید test.py یا test.c و نام شل‌اسکریپت باید script.sh باشد.

۲. ماژول شما به صورت اتوماتیک تحت چندین تست ارزشیابی میشود.

۳. اپلیکیشن تست شما به صورت دستی و همچنین به هنگام تحویل آنلاین ارزشیابی خواهد شد.

۴. فایل‌ها را در پوشه‌ای به نام studentid_prj2 قرار دهید و کمک دستور زیر آنرا زیپ کنید.

```
tar zcf studentid_prj2.tgz studentid_prj2
```

تنها فایل studentid_prj2.tgz را در سامانه یکتا در قسمت مربوط به پروژه دوم بارگذاری کنید.

موفق باشید