دانشگاه صنعتی امیرکبیر

(پلی‌تکنیک تهران)

دانشکده مهندسی کامپیوتر

دستورکار آزمایشگاه سیستم‌های عامل

مسئول آزمایشگاه:

دکتر حمیدرضا زرندی

نسخه ۳ (آزمایشی)

تابستان ۱۴۰۳

بسمه‌تعالی

در برنامه آموزشی دانشکده، این آزمایشگاه هم نیاز درس اصلی سیستم‌های عامل است. لذا اگر دانشجو این درس را درگذشته گذرانده باشد، لازم است مباحث آن مرور گردد. در طول‌ترم برای انجام آزمایش‌ها از سیستم‌عامل لینوکس استفاده می‌شود چنانچه دانشجویی از لپ‌تاپ شخصی در آزمایشگاه استفاده می‌کند، لازم است سیستم‌عامل لینوکس را نصب کند. جهت نصب این سیستم‌عامل، توضیحاتی توسط مدرس ارائه خواهد شد.

- تعداد آزمایش‌های که در طول‌ترم انجام می‌شود، در جلسه اول، توسط مدرس آزمایشگاه تعیین می‌شود، دانشجویان در هر جلسه به‌صورت تک نفره یا دونفره این آزمایش‌ها را انجام می‌دهند.

- زمان اتمام هر آزمایش، توسط مدرس آزمایشگاه با توجه به محتوای آزمایش مشخص می‌شود و دانشجویان قبل از شروع هر آزمایش نسبت به مهلت انجام آن مطلع می‌شوند.

- برای این درس در انتهای ترم، امتحانی در نظر گرفته نشده است، اما طبق صلاحدید مدرس، ممکن است پروژه‌ای مدنظر قرار گیرد.

- قبل از شروع آزمایش هر گروه لازم است پیش گزارش تهیه‌کرده و قبل از شروع کلاس تحویل مدرس آزمایشگاه دهد. مدرس آزمایشگاه قبل از شروع هر کلاس ممکن است پرسش‌های شفاهی یا کتبی نسبت به آزمایش موردنظر مطرح نماید و دانشجویان موظف به پاسخگویی کامل و صحیح هستند.

- ازآنجایی‌که شروع کلاس‌های آزمایشگاه پس از زمان حذف و اضافه است، تعداد جلسات برگزارشده کمتر بوده و لذا حضور در کلیه جلسات الزامی است و تنها یک جلسه غیبت مجاز خواهد بود. همچنین از ورود افراد بیش از ۱۰ دقیقه تأخیر ممانعت به عمل می‌آید.

- نمره دهی نهایی بر اساس موارد زیر انجام خواهد شد (مدرسین آزمایشگاه در صورت لزوم می‌توانند تغییراتی ایجاد نمایند):

پیش گزارش‌های تحویل داده‌شده	مجموع آزمایش‌ها حدود ۱۰ درصد
نمره پرسش‌های شفاهی/کتبی قبل از شروع هر آزمایش	مجموع آزمایش‌ها حدود ۱۵ درصد
انجام کامل هر آزمایش	مجموع آزمایش‌ها حدود ۳۰ درصد
کیفیت انجام هر آزمایش و پیاده‌سازی آن	مجموع آزمایش‌ها حدود ۳۰ درصد
حضور فعال، مؤثر در گروه همکاری با مدرس (کار کلاسی)	حدود ۱۵ درصد
موارد دیگر (با صلاحدید مدرس آزمایشگاه)	به انتخاب مدرس آزمایشگاه

****توجه****

دستور کار آزمایشگاه سیستم عامل نسخه ۳ (آزمایشی)، نسخه به‌روز شده از دستورکارهای قبلی است که به همت برخی مدرسین ترم‌های قبل و دانشجویان کارشناسی سالهای انتهایی کارشناسی دانشکده و به سرپرستی سرکار خانم آفرین، در طول مدت بهار و تابستان ۱۴۰۳ زیر نظر اینجانب تهیه و تدوین شده است. در این دستور کار، سعی شده است ایرادات دستورکارهای قبلی مرتفع و برخی آزمایشات جدید جهت رفع کاستی‌ها ارایه گردد.

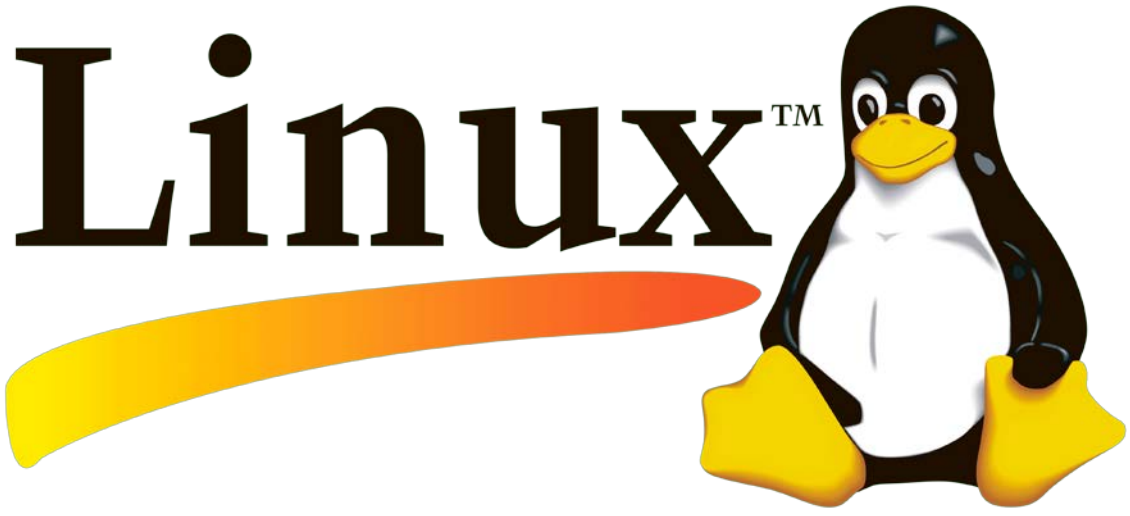
از آنجا که در نسخه اولیه این دستورکار ممکن است همچنان برخی اشکالات ظاهری یا معنایی وجود داشته باشد، بدین‌وسیله از کلیه مدرسین و دانشجویان آزمایشگاه سیستم‌های عامل، درخواست می‌گردد در طول ترم مهر ۱۴۰۳، این دستورکار را به دقت مطالعه، بررسی و اجرا نمایند و در پایان هر آزمایش (یا در پایان کلیه آزمایشات)، اشکالات/ایرادات وارده را به نحو مناسب به اینجانب h_zarandi@aut.ac.ir **یا سرکار خانم آفرین** afaringhazal@gmail.com **یا آقای مهندس کیخا** keikha@aut.ac.ir **منتقل نمایند.**

باتشکر

حمیدرضا زرندی

عضو هیئت علمی دانشکده مهندسی کامپیوتر

آزمایش اول:
آشنایی با لینوکس



هدف آزمایش

آشنا شدن با محیط سیستم عامل
لینوکس

آمادگی پیش از آزمایش:

-

شرح آزمایش

بخش اول: تاریخچه

در سال ۱۹۷۱، سیستم عامل یونیکس (Unix) به دست تعدادی از مهندسان شرکت تلفن و تلگراف آمریکا (AT&T Corp.) توسعه پیدا کرد. این سیستم عامل که هرساله پیشرفته‌تر می‌شد، چندان ارزان نبود و همه نمی‌توانستند از آن استفاده کنند. در سال ۱۹۸۴ میلادی، ریچارد استالمن (Richard Stallman) که رئیس بنیاد نرم‌افزارهای آزاد بود، پروژه «گنو» (GNU) را آغاز کرد. در این پروژه که یک جنبش نرم‌افزاری محسوب می‌شد، برنامه‌نویسان با یکدیگر همکاری می‌کردند که این همکاری تابه‌حال هم ادامه دارد. تا چند سال بعد، ابزارهای متنوعی در پروژه گنو توسعه پیدا کردند. اما این ابزارها برای اجرا، نیازمند یک هسته مناسب و آزاد به عنوان سیستم‌عامل بودند، هسته ای که توسعه آن به این زودی‌ها امکان‌پذیر نبود.

سال ۱۹۹۱، لینوس توروالدز (Linus Torvalds) یک دانشجوی ۲۱ ساله بود که در دانشگاه هلسینکی درس می‌خواند. او در ابتدای این سال، یک کامپیوتر IBM خرید که با سیستم‌عامل MS-DOS کار می‌کرد. او که از این سیستم‌عامل راضی نبود، علاقه داشت که از یونیکس استفاده کند. ولی متوجه شد که ارزان‌ترین نوع سیستم‌عامل یونیکس، ۵ هزار دلار قیمت دارد. به همین خاطر و به دلیل عملکرد ضعیف پروژه گنو در زمینه توسعه هسته سیستم‌عامل، لینوس تصمیم گرفت که خودش دست‌به‌کار شود.

در ۲۵ آگوست همان سال، «لینوس» متنی را به گروه خبری comp.os.minix مبنی بر توسعه هسته یک سیستم‌عامل جدید می‌فرستد و از برنامه‌نویسان می‌خواهد که در این مسیر به او کمک کنند. این‌گونه بود که او اولین نسخه از سیستم‌عامل لینوکس را سپتامبر همان سال منتشر کرد. دومین نسخه آن به فاصله کمی‌در اکتبر همان سال منتشر شد. از آن زمان و تا امروز، هزاران برنامه‌نویس در توسعه لینوکس مشارکت داشته‌اند که به تعداد آن‌ها همواره افزوده می‌شود. اما شاید برخی بپرسند که درنهایت لینوکس هسته سیستم‌عامل است یا به‌تنهایی یک سیستم‌عامل محسوب می‌شود؟

لینوکس چیست؟

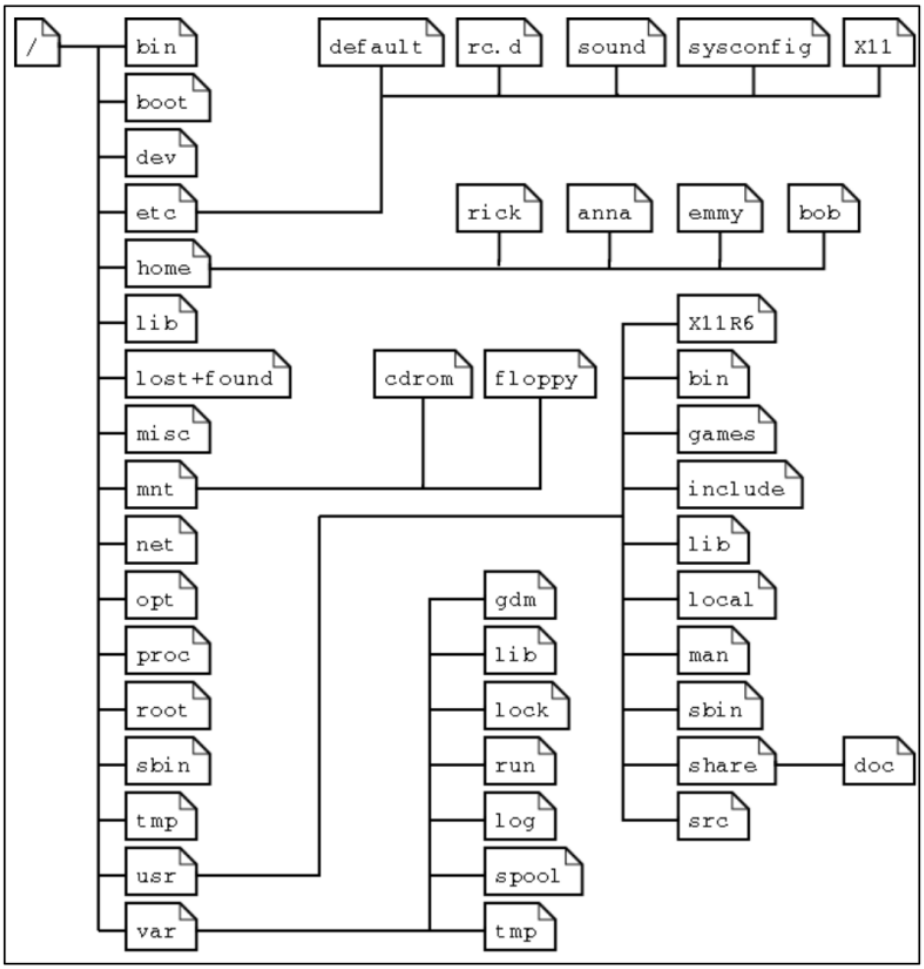
از دید فنی، لینوکس تنها نامی‌است برای هسته سیستم‌عامل و نه کل آن. دلیل این تعریف‌های گوناگون از لینوکس، به دلیل ماهیت انعطاف‌پذیر آن است. کمی‌بعد از عرضه این سیستم‌عامل، توروالدز تصمیم گرفت که به پروژه گنو بپیوندد. با

این کار به‌سرعت توسعه لینوکس افزوده شد و توزیع‌های مختلفی ظاهر شدند. توزیع‌ها مجموعه‌ای از ابزارها هستند که برای رسیدن به اهداف مختلف در کنار هم قرار می‌گیرند و از هسته لینوکس استفاده می‌کنند. به همین خاطر، لغت لینوکس را به سیستم‌عامل‌هایی اطلاق می‌کنند که از ترکیب‌بندی لینوکس (به عنوان هسته سیستم‌عامل) با نرم‌افزارهای آزاد و متن‌باز به دست می‌آیند. درصورتی که بنیاد نرم‌افزارهای آزاد تاکید دارد که از چنین سیستم‌عامل‌هایی، با عنوان گنو/لینوکس یاد شود. در این میان سوالی که برای خیلی‌ها مطرح می‌شود این است که اگر لینوکس متن‌باز و رایگان است، پس درآمد توسعه‌دهندگان توزیع‌های آن چطور به دست می‌آید؟

بخش دوم: نصب سیستم‌عامل لینوکس

در این بخش، دانشجویان باید بتوانند با توضیحات استاد محترم آزمایشگاه، نحوه نصب یک نسخه به‌روز از سیستم‌عامل لینوکس را روی یک ماشین مجازی (یا حقیقی) را یاد گرفته و به‌صورت عملی انجام دهند.

بخش سوم: فایل سیستم لینوکس



دایرکتوری root با / مشخص می‌شود و تمامی‌فایل‌های دیگر را درون خود دارد.

برای شروع این بخش لازم است پایانه (Terminal) را باز کنید (برای این کار می‌توانید از کلیدهای میانبر Ctrl+Alt+T استفاده کنید).

۱. دستور ls برای لیست کردن فایل‌ها و دایرکتوری‌ها استفاده می‌شود. البته می‌توان از سوییچ‌های مختلفی برای این دستور استفاده کرد که هرکدام کار خاص خود را انجام می‌دهند. در زیر لیست سوییچ‌ها قابل‌مشاهده است.

لیست سوییچ‌های دستور ls	
-l	نشان دادن جزئیات بیشتر در لیست
-1	در هر خط فقط یک فایل لیست شود
-t	بر اساس زمان تغییر یافتن مرتب می شود و آخرین تغییر در اول می آید
-r	برعکس کردن اصل مرتب سازی
-s	برای چاپ میزان حافظه مصرف شده برای هر فایل
-R	زیر دایرکتوری‌ها را به‌صورت بازگشتی لیست کند

۲. دستور cp برای کپی کردن فایل‌ها و دایرکتوری‌ها استفاده می‌شود. حالت کلی استفاده در ذیل آمده است:

cp [options] source destination

۳. دستور mv برای جابه‌جا کردن یک فایل و یا دایرکتوری و همچنین برای تغییر نام آن‌ها به کار می‌رود. حالت کلی در ذیل آمده است:

mv [options] source destination

لیست سوییچ‌های دستورهای cp و mv	
-f	قبل از رونویسی بر فایل مقصد، از کاربر چیزی نمی‌پرسد
-i	اگر فایلی با اسم مقصد وجود داشته باشد، از کاربر می‌پرسد که رونویسی کند یا خیر
-b	بر اساس زمان تغییر یافتن مرتب می شود و آخرین تغییر در اول می آید
-p	صفات را حفظ می کند

۴. دستور rm برای پاک کردن یک فایل و یا دایرکتوری به کار می‌رود. حالت کلی در ذیل آمده است:

rm [options] file

لیست سوییچ‌های دستور rm	
-r,-R	برای پاک کردن دایرکتوری‌ها و محتوای داخل آن‌ها به‌صورت بازگشتی
-f	حذف کردن به صورت اجباری
-i	قبل از هر حذف از کاربر سؤال می کند

۵. دستور mkdir برای ساختن دایرکتوری‌ها به کار می‌رود. حالت کلی در ذیل آمده است.

mkdir [options] dir_name

۶. دستور rmdir برای پاک کردن دایرکتوری خالی به کار می‌رود. حالت کلی در ذیل آمده است.

rmdir [options] dir_name

۷. علائم (wildcard) را می‌توان برای استفاده‌های متعددی که در یک مرحله کاربر روی تعداد زیادی فایل می‌خواهد انجام شود استفاده کرد. برای مثال:

دستور	کاری که انجام می‌دهد
rm *	تمام فایل‌های دایرکتوری فعلی را پاک می‌کند
cp * directory	تمام فایل‌های دایرکتوری فعلی را به directory منتقل می‌کند
cp *[a-f] directory	تمام فایل‌های دایرکتوری فعلی که با حروف a تا f تمام میشوند را به directory منتقل می‌کند
ls n*	فایل‌ها و دایرکتوری‌هایی که درون دایرکتوری‌هایی قرار دارند که اسمشان با n شروع میشود را لیست می‌کند
ls t?	فایل‌ها و دایرکتوری‌هایی که درون دایرکتوری‌هایی قرار دارند که اسمشان با t شروع می‌شود و اسمشان دو حرفی است را لیست می‌کند

۸. دستور touch برای تغییر دادن تاریخ و زمان (Timestamp) به کار می‌رود و اگر فایل موجود نباشد آن را ایجاد می‌کند. حالت کلی در ذیل آمده است.

touch [options] file

لیست سوییچ‌های دستور touch	
-a	فقط زمان دستیابی (Access time) تغییر کند
-c	اگر فایلی با این اسم موجود نبود، فایل جدیدی تولید نکند
-d	رشته ای که پس از آن می‌آید را پارس کرده و به‌جای زمان فعلی استفاده می‌کند
-m	فقط زمان تغییر (Modification time) تغییر کند
-r	از زمان‌های فایل به‌جای زمان فعلی استفاده کند
-t	فایلی با زمان مشخص شده تولید کند

سه نوع از تاریخ و زمان در زیر شرح داده شده است:

۱. Access time: آخرین زمانی است که فایل خوانده شده است.

۲. Modification time: آخرین زمانی که محتوای فایل تغییر کرده است.

۳. Change time: آخرین زمانی که ابر داده ی (Metadata) فایل (مانند Permission) تغییر کرده است. برای مثال استفاده‌های مختلفی از این دستور در جدول زیر قابل‌مشاهده است:

```
touch filename
touch -d 10am filename
touch -d 13:50 filename
touch -d “yesterday 9pm” filename
touch -r reference_file target_file
```

مثال: برای مشاهده زمان دستیابی (Access time) فایل‌ها از ls -l استفاده کنید.

۹. دستور find برای جست‌وجوی سلسله مراتبی استفاده می‌شود:

لیست سوییچ‌های دستور find	
-name	دنبال الگویی که پس از این سوییچ می‌آید می‌گردد
-iname	فرقی با بخش بالایی ندارد به‌جز اینکه به کوچک یا بزرگ بودن حروف حساس نیست
-type d	جست‌وجوی دایرکتوری
-type f	جست‌وجوی فایل

-size +N/-N	برای جست‌وجو براساس حجم فایل استفاده می‌شود. + به معنی بزرگ‌تر از N و - به معنی کوچک‌تر از N است. اگر عدد خالی بیاید به معنی بلوک است و با استفاده از c برای کاراکتر، G برای گیگابایت و... می‌توان حجم را معلوم کرد.
-empty	برای جست‌وجوی فایل یا دایرکتوری خالی استفاده می‌شود.
-atime n	برای جست‌وجوی فایل‌هایی که با n*24 ساعت قبل خوانده شده است.
-ctime n	برای جست‌وجوی فایل‌هایی که با n*24 ساعت قبل metadata آن تغییر کرده است.
-mtime n	برای جست‌وجوی فایل‌هایی که n*24 ساعت قبل محتوای آن تغییر کرده است.
-amin n	برای جست‌وجوی فایل‌هایی که با n دقیقه قبل خوانده شده است.
-cmin n	برای جست‌وجوی فایل‌هایی که با n دقیقه قبل metadata آن تغییر کرده است.
-mmin n	برای جست‌وجوی فایل‌هایی که n دقیقه قبل محتوای آن تغییر کرده است.

مثال:

```
find .
find directory/
find . -name “f*”
find . -iname “f*”
find . -type f -iname “t*”
find . -type d -iname “t*”
find -size 65G
find . -size +5k
find -empty
find . -mtime -1
find . -amin -45 -type d
```

نکته: اگر قرار باشد روی فایل‌هایی که با استفاده از دستور بالا پیدا شده است، عملی انجام شود، راه مناسب استفاده از exec است که پس از این از سوییچ {} یا '{}' برای اشاره به فایل‌ها و پس از پایان دستور از \; باید استفاده کرد.

مثال:

```
find . -mmin -1 -exec cat ‘{}’ \;
find /etc/rc* -exec echo Arg: {} \;
```

۱۰. از دستور file برای مشاهده نوع فایل به‌طوری که برای بیننده واضح باشد می‌توان استفاده کرد.

۱۱. دستور gzip و gunzip برای فشرده‌سازی و باز کردن فایل فشرده استفاده می‌شود. این دستورات پس از فشرده سازی، نسخه اصلی را پاک می‌کنند و فایل جدید با اسم قبلی ولی با پسوند ".gz" می‌سازند. با استفاده از پسوند -d می‌توان فایل فشرده را باز کرد. مثال:

```
gzip filename
gzip -d filename.gz (Decompress.)
gunzip filename.gz
```

بخش پنجم: مالکیت و مجوزهای فایل

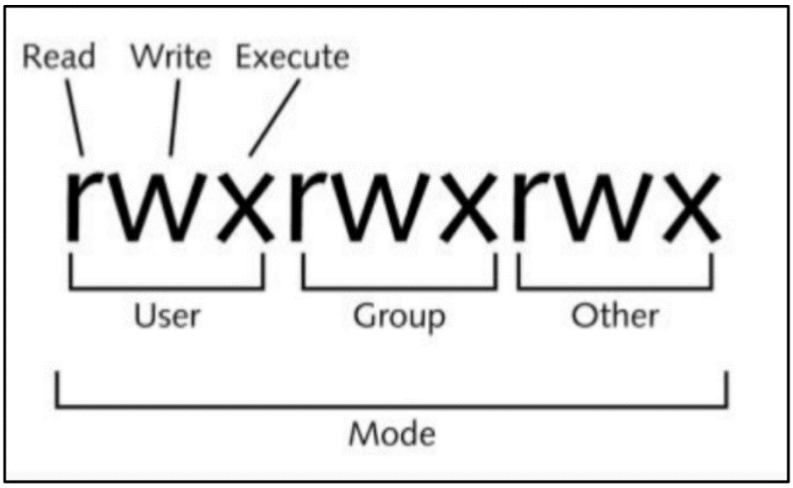
هر فایل شامل سه قسمت مجوز است:

الف) User permission: مربوط به مالک فایل است.

ب) Group permission: مربوط به گروه‌های تعریف شده در سیستم است.

ج) Other permission: مربوط به سایر افراد استفاده کننده از سیستم است.

هر دسته می‌تواند نوع مجوزهای خاص خود را داشته باشند.



۱. دستور chown برای تغییر مالکیت فایل و دایرکتوری استفاده می‌شود. فقط کاربر اصلی می‌تواند این کار را در لینوکس انجام دهد. مثال:

```
sudo chown root:root hello.sh
```

۲. دستور chgrp برای تغییر مالکیت گروهی فایل و دایرکتوری استفاده می‌شود. فقط کاربر اصلی می‌تواند این کار را در لینوکس انجام دهد. مثال:

```
chgrp adm hello.sh
```

۳. دستور chmod برای تغییر اجازه‌ها برای فایل و دایرکتوری استفاده می‌شود. حالت کلی در ذیل آمده است.

chmod symbolic-mode filename

دسته‌بندی‌هایی که با آن‌ها کار می‌شود:

1. u = user
2. g = group
3. a = all

عملیات:

1. set (=)
2. remove (-)
3. give (+)

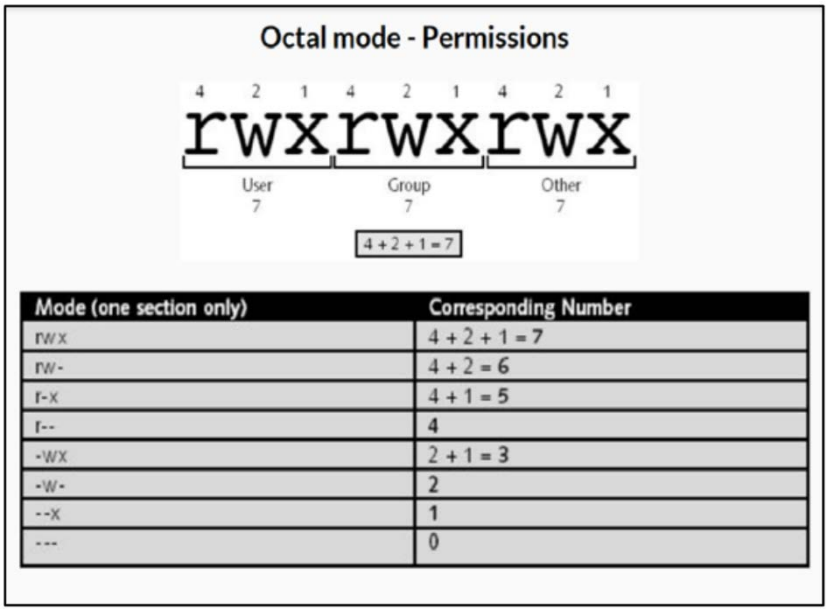
مجوزها:

1. read (r)
2. write (w)
3. execute (x)

مثال:

```
chmod u+x filename
chmod ug-x filename
chmod o-r filename
chmod o=wrx filename
chmod o=r,g=r,u=wrx filename
```

همچنین مدل دیگری برای این کار وجود که در شکل زیر قابل‌مشاهده است:



Mode (one section only)	Corresponding Number
rwx	4 + 2 + 1 = 7
rw-	4 + 2 = 6
r-x	4 + 1 = 5
r--	4
-wx	2 + 1 = 3
-w-	2
--x	1
---	0

مثال:

```
chmod 755 filename
755: rwxr-xr-x
744: rwxr--r--
777: rwxrwxrwx
666: rw-rw-rw-
```

بخش ششم: خواندن محتوای فایل‌ها

یک فایل با محتوای متنی کوتاه (۵ الی ۱۵ خط نوشته) را انتخاب کنید. دستور cat را به‌صورت زیر روی آن صدا بزنید:

cat filename

با این دستور محتوای فایل در terminal چاپ می‌شود. اگر فایل متنی طولانی تر باشد خواندن آن به شدت سخت‌تر می‌شود. دستور دیگری برای خواندن محتوای متنی به نام less وجود دارد. آن را به‌صورت زیر روی فایل صدا بزنید:

less filename

با استفاده از این دستور، دیگر کل محتوای فایل چاپ نمی‌شود، بلکه بخش بالایی آن مشاهده می‌شود و امکان جابه‌جایی در فایل داریم. در محیط less می‌توان از دستورات زیر برای جابه‌جایی در فایل و کارهای دیگر استفاده کرد:

لیست دستورهای less	
Down arrow, Enter, e, j	در محتوای فایل یک خط به پایین می‌رود
Up arrow, y, k	در محتوای فایل یک خط بالا می‌رود
Space bar, f	یک صفحه در محتوای فایل به پایین می‌رود
b	یک صفحه در محتوای فایل به بالا می‌رود
g	به اولین خط فایل می‌رود
Ng	به N امین خط فایل می‌رود
G	به آخرین خط فایل می‌رود
q	از محیط less خارج می‌شود

بخش هفتم: خواندن راهنمای دستورها

دستورات terminal لینوکس برای استفاده، راهنماهایی به اسم manual page (به اختصار man) دارند که به کاربر کمک می‌کنند خیلی خلاصه و سریع استفاده از دستور را یاد بگیرد و بتواند از آن استفاده کند.

در manual page معمولا یک توضیح کوتاه، یک توضیح بلندتر و بعد switchها و کاربردهای آن‌ها و مثال‌هایی از استفاده آن‌ها نمایش داده می‌شوند. دستور زیر را در terminal وارد کنید:

```
man ls
```

با دستور man، راهنمای دستور مورد نظر (که در این مثال ls است) در محیط less به کاربر نمایش داده می‌شود. اگر درمورد man اطلاعات بیشتری نیاز داشتید، می‌توانید راهنمای آن را با دستور man man مطالعه کنید.

تمرین‌ها: (خروجی مورد انتظار آزمایش)

۱. دایرکتوری داخل میز کاری (Desktop) بسازید و تمامی‌مجوزهای آن را به گونه‌ای تغییر دهید که فقط شما و اعضای گروه بتوانند بنویسند، بخوانند و در آن جست‌وجو کنند.
۲. گروه‌هایی که شما در آن عضو هستید، را لیست کنید، سپس مالکیت فایل قبلی را به یکی دیگر از گروه‌های خود بدهید.
۳. این دستور چه‌کاری انجام می‌دهد؟

```
chmod 4664 file.txt
```

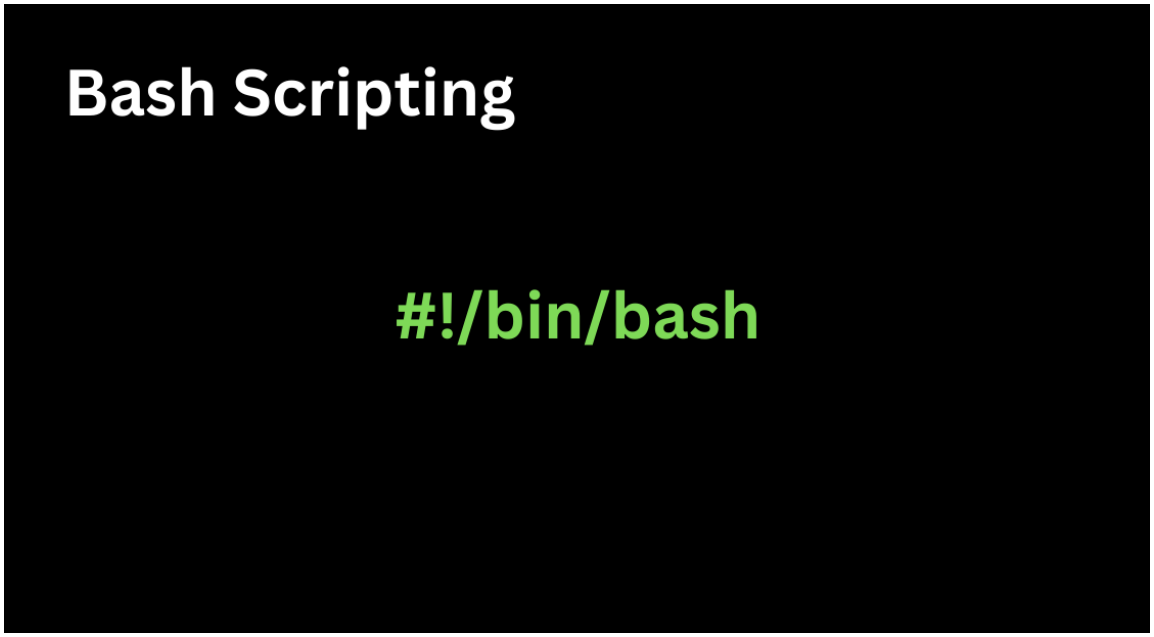
۴.درون کل دایرکتوری‌های موجود، فایل‌های خالی را پیدا کرده و پاک کنید (این کار باید در یک خط دستور انجام شود).
۵. با مطالعه manual page دستور less، راهی برای search کردن یک عبارت درون متن یک فایل پیدا کنید.

مراجع مطالعه/پیوست‌ها:

1. R. Smith, LPIC-1, 3rd ed. Indianapolis, Indiana: John Wiley & Sons, 2013
2. https://jadi.gitbooks.io/lpic1/content

آزمایش دوم:

دستور نویسی در سیستم عامل



هدف آزمایش

آشنایی با دستور نویسی در سیستم
عامل و خودکارسازی کارهای لازم در
خط دستور

آمادگی پیش از آزمایش:

آشنایی با فایل سیستم های لینوکس و دستورات
مقدماتی ترمینال لینوکس

شرح آزمایش

مقدمه:

به شکل خلاصه Bash یک مفسر زبان دستوری (Command Language Interpreter) برای کار با سیستم عامل از طریق ترمینال یا خط فرمان (Command Line) است که به آن پوسته یا Shell نیز می‌گویند. اسم پوسته به این دلیل برایش انتخاب شده که مانند یک پوسته هسته‌ی سیستم عامل را در بر می‌گیرد و به ما اجازه می‌دهد که دستورات و کارهای مهمی را که با سیستم عامل داریم، بدون نیاز به توجه به جزئیات و دستکاری هسته انجام دهیم. علاوه بر این‌ها خود bash یک زبان برنامه‌نویسی نیز هست که به ما اجازه می‌دهد که برای اجرای دستورات کدهایی در پوسته بنویسیم و اجرا کنیم. این دستورات جدید همانند دستورات سیستم در نشانی‌هایی مانند /bin هستند و این امکان را ایجاد می کنند تا کاربران یا گروه ها محیط هایی شخصی را برای بهینه سازی کارهای معمول خود ایجاد کنند.

با نوشتن مجموعه ای از دستورات در یک فایل متنی می توان‌به جای اجرای تک تک‌دستورات در ترمینال لینوکس، با اجرای فایل به اجاری همه‌ی آن‌ها با ترتیب مشخص شده دست یافت. اگرقالب این فایل به صورت sh. باشد و اسم فایل را sample.sh فرض کنیم (در لینوکس پسوند فایل‌ها اهمیتی ندارد و آنچه نوع فایل را مشخص می‌کند magic numberها یا همان چند بایت ابتدایی فایل هستند)، می‌توان با دستور /sample.sh. این فایل را اجرا کرد.حالت دیگر آن است که از دستور bash filename در ترمینال استفاده شود. لازم است در خط اول فایل عبارت /#!bin/bash قرار گیرد تا مشخص شود مفسر این دستورات bash است. به علامت !# شبنگ)Shebang(می‌گویند.

همانطور که می‌دانید، همه چیز در یونیکس یک فایل است و یونیکس بین فایل‌ها مکانیزمی به نام جریان (stream) را تعیین می کند که به داده ها اجازه می دهد بیت به بیت از یک فایل به یک فایل دیگر حرکت کنند. جریان دقیقا چیزی است که به نظر می رسد: یک رودخانه کوچک از بیت ها که از یک فایل به دیگری می‌ریزد. اگرچه شاید پل نام بهتری باشد زیرا بر خلاف جریان، که یک جریان دائمی از آب است، جریان بیت‌ها بین فایل‌ها نباید ثابت باشد یا حتی لزوما استفاده شود.

سه جریان استاندارد پایه برای همه ی فایل‌ها وجود دارد:

- Standard in (stdin): جریان استاندارد برای ورودی به فایل
- Standard out (stdout): جریان استاندارد برای خروجی از فایل
- Standard error (stderr): جریان استاندارد برای خطاهای خروجی از فایل

در این قسمت می‌خواهیم با redirection و piping آشنا شویم:

1. Process > data file:

خروجی فرآیند را به فایل داده هدایت می‌کند؛ فایل را در صورت لزوم ایجاد می‌کند، در غیر این صورت فایل‌های موجود را بازنویسی می‌کند.

2. Process >> data file:

خروجی فرآیند را به فایل داده هدایت می‌کند؛ فایل را در صورت لزوم ایجاد می‌کند، در غیر این صورت به محتوای موجود آن اضافه می‌کند.

3. Process < data file:

محتویات فایل داده را می‌خواند و آن را به ورودی فرآیند هدایت می‌کند.

4. Process_1 | Process_2 (pipping):

خروجی Process_1 (stdout) را به‌عنوان ورودی برای Process_2 (stdin) ارسال می‌کند. برای هر کدام از جریان‌هایی که در بالا ذکر شد، مثال می‌زنیم که بیشتر با آن آشنا شویم.

برای مثال فرض کنیم کدی را روی سیستم اجرا کردیم که در هر دفعه اجرا شدن لاگ (Log)های خودش را در stdout می‌نویسد. برای اینکه این نوشتار را به فایلی منتقل کنیم که در هر زمان قابل خواندن باشد می‌توانیم از دستوری که در بخش اول نام برده استفاده کنیم.

بخش دوم مشابه بخش اول کار می‌کند اما یک تفاوت جزئی دارد. در بخش قبلی در صورتی که فایلی با همان نام مد نظرمان از قبل وجود داشته باشد، اطلاعات داخل فایل قبلی کامل پاک می‌شود یا به اصطلاح فایل override می‌شود. اما در صورت از استفاده از علامت >> در صورت اجرای کد دیتای جدید نوشته شده در stdout به انتهای فایل قبلی ضمیمه (Append) می‌شود، یا در صورت عدم وجود فایل، این فایل ایجاد می‌شود.

بخش سوم به این صورت است که پردازه ی ما اطلاعات فایل data را به عنوان stdin خود دریافت می‌کند. بر اساس می‌توانیم تصمیم بگیریم که چه اقدامی‌انجام دهیم.

نهایت در pipeline یا خط لوله هر دو طرف، پردازه هستند و خروجی پردازه اول به عنوان ورودی به پروسه دوم پاس داده می‌شود.

چگونه در ترمینال کد بنویسیم؟

. nano:

دستور nano یکی از ساده‌ترین و کاربردی‌ترین دستورهایی است که در کد نویسی در محیط کنسول استفاده می‌شود. بعد از اجرای دستور با چنین محیطی رو به رو می‌شویم:

در محیط nano برخلاف برخی از ابزارهای دیگر نیاز به حفظ کردن هیچ کلید میانبری (Shortcut) برای انجام دستورهای اولیه وجود ندارد. برای یادگیری کلیدها و میانبرهای این ویرایشگر می‌توانید از دستور man که نحوه استفاده از آن در دستور کار اول توضیح داده شده استفاده کنید.

. vim:

ویم نسخه حرفه‌تر ویرایشگر متن موجود در سیستم عامل لینوکس است، یادگیری اولیه vim بخاطر میانبرهایی که دارد ممکن است سخت‌تر از یادگیری nano باشد ولی بخاطر قابلیت‌های آن، بسیاری از کاربران حرفه ای لینوکس استفاده از vim را ترجیح می‌دهند. برای یادگیری بیشتر می‌توانید از لینک زیر کمک بگیرید:
https://github.com/iggredible/Learn-Vim?tab=readme-ov-file

مقداردهی‌به متغیرها:

مانند هر زبان برنامه‌نویسی دیگر در این زبان هم می‌توان متغیر تعریف کرد و به آن مقادیری نسبت داد. به مثال‌های زیر دقت کنید و برای هر قسمت فایلی ایجاد کرده و آن را اجرا کنید. در لینوکس برای چاپ مقادیر در کنسول از دستور echo استفاده می‌شود.

```
#!/bin/bash

#variable assignment

# no space around = during assignment

a=24
echo $a
```

```
echo "$a"
echo "The value of \"a\" is $a."

a=`echo Hello!            # Assigns result of 'echo' command to 'a'

echo $a

a=`ls -l                  # Assigns result of 'ls -l' commandto 'a'
echo "$a"

echo $a                   # Unquoted, however, it removes tabs and newlines.

# Assignment using 'let'

let a=16+5

echo "The value of a is now $a."
```

متغیرهای خاصی وجود دارند که مقادیر آنها از قبل تعیین شده اند و می توان‌در کاربردهای خاص از آنها استفاده کرد. مانند:
$USER, $0, $1 تا $9, $#, $$, $@

که در هنگام اجرای فایل می‌توان آن‌ها را به عنوان آرگومان ورودی به فایل داد. مثال:

```
 bash sample_file 1 3
```

در آن مقادیر ۱ و ۳ که با فاصله آمده اند آرگومان هستند و برای استفاده از این آرگومان ها باید از متغیرهای $1 و $2 استفاده کرد.

سوال:

مقدار سایر متغیرهای خاص را بیابید. اگر بیش از ۱۰ آرگومان ورودی داشته باشیم، چگونه باید به ۱۰ امین آرگومان دست یافت؟

برای دریافت مقادیر مورد نیاز از کاربر در حین اجرای برنامه از دستور read استفاده می شود .به مثال زیر توجه کنید.
p- و sp- چه امکانی را فراهم می کنند؟ کد را اجرا کنید و مقدار متغیرهای uservar و passvar را در فایلی ذخیره کنید.

```
read -p 'Username: ' uservar
read -sp 'Password: ' passvar
```

برای انجام محاسبات شیوه های مختلفی وجود دارد. به مثال های زیر توجه کنید. کد را اجرا کنید و نتیجه را گزارش کنید.

```
let a=10+8
echo $a

expr 5 \* 4
expr 5/4
```

```
expr 11% 2

a=$(expr 10-3)
echo $a

[b=\$((a+3))]
echo $b
((b++))
echo $b
```

عبارت شرطی:

برای نوشتن شرط از قالب زیر پیروی کنید :

```
if []; then
   command1
elif []; then
   command2
else
   command3
fi
```

نکته) اگر چند شرط متفاوت داشته باشیم می‌توان از آنها به اینگونه استفاده کرد: [ ] || [ ] یا [ ] && [ ]

نکته) برای مقایسه اعداد می توان‌از -ls و -gt و -eq استفاده کرد که در مثال زیر به کار رفته است:

```
var1=10
var2=20

if [ $var2 -gt $var1 ]; then
   echo "$var1 is greater than $var2"
fi
```

داخل [ ] بین عبارت با "[" و "]" که به آن دستور تست می‌گویند یک فاصله قرار می‌گیرد.

عبارت چند حالته:

```
case $variable in
   pattern-1)
       commands
       ;;
   pattern-2)
       commands
       ;;
   pattern-3|pattern-4|pattern-5)
       commands
       ;;
   pattern-N)
       commands
       ;;case $variable in
```

```
   pattern-1)
       commands
       ;;
   pattern-2)
       commands
       ;;
   pattern-3|pattern-4|pattern-5)
       commands
       ;;
   pattern-N)
       commands
       ;;
   *)
       commands
       ;;
esac
```

برخی مفاهیم وجود دارند که به شما در نوشتن کد کمک می‌کنند:

#	($#) Expands to the number of positional parameters in decimal.
?	($?) Expands to the exit status of the most recently executed foreground pipeline.
-	($-, a hyphen.) Expands to the current option flags as specified upon invocation, by the set builtin command, or those set by the shell itself (such as the -i option).
$	($$) Expands to the process ID of the shell. In a () subshell, it expands to the process ID of the invoking shell, not the subshell.
-n STRING	The length of STRING is greater than zero
. -z STRING	The lengh of STRING is zero (ie it is empty).
-d FILE	FILE exists and is a directory.
-e FILE	FILE exists.
-r FILE	FILE exists and the read permission is granted.
-s FILE	FILE exists and it's size is greater than zero (ie. it is not empty).
-w FILE	FILE exists and the write permission is granted.
-x FILE	FILE exists and the execute permission is granted.

حلقه‌ها:

```
while [ condition ]
do
   command1
   command2
   command3
done

#! /bin/bash

counter=0
while [ $counter -lt 10 ]
do
   echo "The counter is $counter"
   let counter=counter+1
```

```
done
```

قالب حلقه for و مثالی از آن در ادامه آمده است:

```
for VARIABLE in 1 2 3 4 5 .. N
do
    command1
    command2
    command3
done

for VARIABLE in file1 file2 file3 .. fileN
do
    command1
    command2
    command3
done



for OUTPUT $(Linux-or-Unix-Command-Here)
do
    command1
    command2
    command3
done

for i in $(ls)
do
    echo item: $i
done
```

توابع:

```
function function_name(){
  command1
  command2
  command3
  return 1
}
```

دقت کنید قبل از تعریف تابع نمی‌توان از آن استفاده کرد. اگر در تابع از دستور return استفاده شود مقدار آن توسط $? قابل دسترسی است. برای ارسال آرگومان به تابع مشابه برنامه عمل می‌شود به مثال زیر دقت کنید:

```
function greeting(){
   echo hello $1
   return 2
}
```

```
greeting john
echo $?
```

عبارت $? خروجی آخرین تابع اجرا شده رو در خودش ذخیره می‌کند.

تمرین‌ها (خروجی مورد انتظار آزمایش):

۱. کدی بنویسید که عددی را به عنوان ورودی آرگومان از کاربر دریافت کند. در صورتی که عدد کوچک تر از صفر بود، عبارت 'The weather is freezing' و در صورتی که بین صفر و ۳۰ درجه بود، عبارت 'The weather is cool' و در صورتی که دما بالای ۳۰ درجه بود، عبارت 'The weather is hot' را در ترمینال چاپ کند.

۲. ماشین حسابی با استفاده از case طراحی کنید که به عنوان عدد اول، عدد دوم و علامت ریاضی علامتی که می‌خواهیم بین این دو انجام بشود را به عنوان ورودی دریافت و عنوان خروجی نتیجه را به ما نشان دهد، در صورتی که ورودی‌ها نامعتبر بود یا در انجام عملیات دچار مشکل شدیم نیز خروجی مرتبط به آن نمایش داده شود.

۳. در ادامه تمرین اول اسکریپ اولیه را به گونه ای تکمیل کنید که دمای سانتی گراد وارد شده توسط کاربر را به فارنهایت تبدیل کند و در خروجی نمایش دهد.

۴. کدی بنویسید که ارقام عددی را به شکل متوالی از کاربر دریافت کند، و عددی را چاپ کند که ترتیب ارقامش معکوس عدد اولی باشد. برای مثال در صورت دریافت ارقام به شکل ۶۷۸ عدد ۸۷۶ را در ترمینال چاپ کند.

* تمرین‌های امتیازی (نیازمند مطالعه و جستجو):

۵. کدی بنویسید که آدرس مطلق یا نسبی یک دایرکتوری را به عنوان آرگومان دریافت می‌کند و به عنوان خروجی تعداد فایل‌های موجود در آن دایرکتوری را نمایش می‌دهد.

۶. در رابطه با دستور awk مطالعه کند و یک نمونه از نحوه کار با این دستور و استفاده از در تغییر دادن فایل‌های متنی با رشته‌های طولانی را به دلخواه نشان دهید.

مطالعه بیشتر

در دستور کار اول به شکل اجمالی با لینوکس و سیستم عامل unix آشنا شدیم، حال می‌خواهیم مفهوم distribution یا همان توزیع در سیستم عامل‌های مبتنی برای Linux را بررسی کنیم. به طور خلاصه توزیع‌های لینوکس سیستم عامل‌هایی که هستند که تجمیع هسته linux به همراه باقی نرم افزارهای مورد نیاز برای استفاده از سیستم عامل با توجه به کاربرد مورد نیاز و یک سیستم مدیریت بسته یا به اصطلاح package manager تشکیل می‌شود.

برخی از معروف‌ترین توزیع‌های لینوکس عبارت‌اند از:

.ubuntu (خوانده می‌شود اوبونتو)

اوبونتو معروف‌ترین و پر استفاده‌ترین توزیع لینوکس در دنیا محسوب می‌شود که خود بر پایه توزیع Debian Gnu/linux به وجود آمده است. نسخه عادی اوبونتو که معمولا کاربران حرفه تر و یا از طرفی دیگر کاربران خانگی لینوکس از آن استفاده می‌کنند هر ساله در ماه‌های آوریل و اکتبر عرضه می‌شود. از طرفی دیگرهای نسخه‌های LTS (Long Time Support) اوبونتو وجود دارند که این نسخه‌ها هر دو سال یک بار در آپدیت ماه آوریل عرضه می‌شوند (مثل ۱۸.۰۴ که در حال حاضر روی کامپیوترهای آزمایشگاه نصب هستند).

تفاوت این دو نسخه در این است که نسخه‌های LTS عموما از پایداری بیشتری از نسخه‌های عادی اوبونتو برخوردار هستند و مدت بیشتری توسط بنیاد canonical آپدیت‌های منظم دریافت می‌کنند که آن‌ها را برای کاربردهایی که نیاز به ثبات بیشتری دارند (سرورهای بک‌اند، سیستم‌های مدارس و ...) مناسب تر می‌کند.

.redhat

توزیع‌های مبتنی بر redhat که معروف‌ترین آن‌ها RHEL می‌باشد، توزیع‌هایی از لینوکس هستند که توسط شرکت redhat عرضه می‌شود و در نقطه مقابل سیستم‌های مبتنیبر debian قرار می‌گیرد. این نسخه‌های لینوکس بر خلاف اکثر سیستم‌های مبتنی بر debian پولی هستند. کاربر در ازای مبلغی که برای استفاده از این سیستم عامل پرداخت می‌کند پشتیبانی شرکت redhat را دریافت می‌کند. این موضوع استفاده از RHEL را برای شرکت‌هایی که قصد استفاده از لینوکس برای موارد خاص خود را دارند مناسب می‌کند. سیستم‌های مبتنی بر redhat عموما از نرم افزار مدیریت بسته rpm برای مدیریت بسته‌های خود استفاده می‌کنند.

.kali

اگر به حوزه شبکه علاقه‌مند باشید ممکن است اسم kali به گوشتان خورده باشد. کالی توزیعی از لینوکس بر پایه debian می‌باشد. در کالی حدود ۶۰۰ ابزار مرتبط با تست نفوذ (penetration testing) به شکل پیش فرض نصب هستند (شاید برایتان جالب باشد که بدانید شخصیت Eliot در سریال Mr. Robot از کالی استفاده می‌کرد).

آزمایش سوم:
برنامه‌نویسی ماژول‌های هسته
و آشنایی با ساختمان‌های داده در هسته

هدف آزمایش

آشنا شدن با نحوه نوشتن و اضافه کردن
واحدهای هسته و همچنین استفاده از
ساختمان داده‌های معروف هسته

آمادگی پیش از آزمایش:

پیشنهاد می‌شود که قبل از انجام این آزمایش, برای یادگیری و کدنویسی ساده تر در ویرایشگرها, به نصب هدرهای لینوکس بپردازید تا بتوانید از قابلیت‌های این ویرایشگرها برای مشاهده کدهای کتاب‌خانه‌ها استفاده کنید. در صورتی که پس از نصب, همچنان ویرایشگر شما توانایی شناسایی کتاب‌خانه‌ها را نداشت, می‌توانید به این لینک (برای افزودن دستی آدرس هدرها) مراجعه کنید.

شرح آزمایش

بخش اول: ماژول‌های هسته لینوکس

سیستم عامل لینوکس, قابلیت این را دارد که بتوان به صورت پویا و بدون نیاز به راه‌اندازی مجدد (reboot) سیستم، قطعه کدهای اجرایی به آن اضافه کرد. این قابلیت به لینوکس این امکان را می‌دهد که بتوان راحت‌تر ویژگی‌های مدنظرمان را به هسته اضافه کنیم و همچنین نگهداری و مدیریت آن نیز آسان می‌شود. به عنوان مثال، از کاربردهای آن می‌توان به اضافه کردن درایورهای دستگاه‌ها (مانند usb driver)، ماژول‌های برای اضافه کردن ویژگی‌های جدید به filesystem و یا نوشتن ماژول‌هایی برای مدیریت متفاوت فرایندها (process) اشاره کرد. البته توجه داشته باشید که با توجه به این که این ماژول‌ها با دسترسی هسته (kernel) انجام می‌شوند، هر خطایی در آن‌ها ممکن است باعث خرابی عملکرد سیستم شود تا جایی که نیاز به راه‌اندازی مجدد (reboot) باشد، به همین دلیل پیشنهاد می‌شود که از ماشین مجازی برای این آزمایش استفاده کنید تا بازگشت به حالت اولیه سیستم آسان تر باشد.

در بخش اول این آزمایش قرار است که با نحوه کدنویسی و اضافه کردن این ماژول‌ها به هسته آشنا شویم. اولین دستوری که با آن آشنا می‌شوید، دستور زیر می‌باشد که با آن می‌توانید لیست تمام ماژول‌هایی که اکنون در هسته بارگذاری شده‌اند را مشاهده کنید:

lsmod

این دستور ماژول‌های بارگذاری شده را در سه ستون نام، اندازه و جایی که از آن ماژول استفاده می‌شود لیست می‌کند.

کدنویسی ماژول‌ها:

هر ماژول هسته، باید حداقل دو تابع اصلی داشته باشد. یک تابع برای زمانی که ماژول در هسته اضافه می‌شود و تابع دیگر برای زمانی که ماژول از هسته خارج می‌شود. این دو تابع هر نام دلخواهی می‌توانند داشته باشند (البته باید امضای آنها به فرمت خاصی برای ماژول‌ها باشد) ولی در کد، باید این توابع به عنوان توابع ورودی و خروجی ماژول ثبت می‌شوند. به مثال زیر توجه کنید:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/printk.h>
```

کد زیر برای هنگام ورود ماژول به هسته:

```
/* this function is called when the module is loaded*/
int simple_init(void)
{
pr_info(“Loading module\n”);
// is alias to -> printk(KERN_INFO "Loading Module\n");
return 0;
}
```

و کد زیر برای اجرا در هنگام خروج :

```
Void simple_exit(void)
{
    pr_info(“Removing module\n”);
}
```

و کد زیر برای ثبت این توابع به عنوان نقطه ورود و خروج استفاده می‌شود :

```
module_init(simple_init);
module_exit(simple_exit);
```

و در نهایت می‌توانید اطلاعات دیگری مانند مجوز، نام نویسنده و ... را می‌توانید اضافه کنید:

```
MODULE_LICENCE(“GPL”);
MODULE_DESCRIPTION(“simple module”);
MODULE_AUTHOR(“OS-Lab-Group”);
```

دقت شود که برای پرینت خروجی، از تابع pr_info() استفاده کردیم. این تابع معادل printk در سطح KERN_INFO می‌باشد. این تابع خروجی را در بافر سابقه هسته ذخیره می‌کند. همچنین، می‌توان پیام‌ها را در سطوح دیگری مانند KERN_DEBUG, KERN_WARNING, KERN_ALERT و ... نیز چاپ کرد که می‌توانید به صورت جداگانه بررسی کنید.
برای مشاهده پیام‌های چاپ شده در بافر هسته، می‌توانید از دستور زیر استفاده کنید:

```
Sudo journalctl –since “5 minute ago”
```

این دستور لاگ‌های سیستمی ۵ دقیقه اخیر را در کنسول چاپ می‌کند. همچنین می‌توانید برای محدود کردن نتیجه‌ها از دستور زیر استفاده کنید:

```
Sudo journalctl –since “5 minute ago” | grep “Removing module”
```

کامپایل کد با استفاده از Makefile

پس از نوشتن کد ماژول، باید برای کامپایل این کد از Makefile استفاده کنیم. در این فایل ما نحوه کامپایل و وابستگی‌های ماژول (dependency) را تعیین می‌کنیم و سپس با استفاده از دستور make کدها را کامپایل می‌کنیم.
در فایل Makefile کدی به اینصورت نوشته می‌شود:

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

- یک متغیر (در اینجا obj-m) برای نگهداری لیست شیءهایی که قرار است در این فرایند ساخته شوند استفاده می‌شود. دقت شود که نام این شیءها (بدون پسوند .o) باید برابر با نام فایل کد ماژول باشد.
- در خط بعدی، PWD:= $(CURDIR) یک متغیر تعریف می‌کند که آدرس دایرکتوری فعلی را در آن نگهداری می‌کند.
- در این کد، کد نوشته شده در قسمت all هنگام کامپایل کدها اجرا می‌شود (در هنگام اجرای دستور make در ترمینال) و کد نوشته شده در قسمت clean برای پاک کردن فایل‌های ساخته شده هنگام کامپایل استفاده می‌شود (در هنگام اجرای دستور make clean در ترمینال)
- کد نوشته شده در قسمت all، به این مفهوم است که باید برای اجرای دستور make ابتدا دایرکتوری را عوض کند (-C) و همچنین، M=$(PWD) به این مفهوم است که ماژول‌ها در کجا قرار دارند و در کجا باید به دنبال ماژول‌ها بگردد و در آخر modules هم تگ دیگری است که برای زمان کامپایل باید به دستور make داده شود تا مشخص شود که هدف کامپایل کردن است.
- کد نوشته شده در قسمت clean هم مانند قبلی می‌باشد با این تفاوت که با تگ clean کردن باید اجرا شود.
- دقت شود که برای به دست آوردن آدرسی که باید دستور در آن اجرا شود از دستور uname -r استفاده شده است که نسخه لینوکسی که در حال حاضر در حال اجراست را چاپ می‌کند.
- برای اطلاعات بیشتر درباره نحوه استفاده از Makefile می‌توانید از این لینک استفاده کنید.

پس از ساخت فایل Makefile در دایرکتوری کدهای ماژول‌ها، با دستور make آن را کامپایل کنید تا فایل simple_module.ko را مشاهده کنید. حال برای اضافه کردن این ماژول به هسته از دستور زیر استفاده می‌کنیم:

```
Sudo insmod simpleModule.ko
```

با اضافه کردن این ماژول، تابع simple_init اجرا می‌شود. حال برای خارج کردن آن از هسته دستور زیر را اجرا می‌کنیم:

```
Sudo rmmod simpleModule
```

با اجرای این دستور، تابع simple_exit اجرا می‌شود.

ماژول‌های هسته در سیستم عامل‌های دیگر

علاوه بر لینوکس، سیستم عامل‌های دیگر مانند Windows یا macOS هم سازوکارهایی برای افزودن ماژول به هسته اضافه کرده‌اند که البته در مواردی با لینوکس متفاوت هستند و برنامه‌نویسی و افزودن ماژول را دشوارتر می‌کنند.

به عنوان مثال، در سیستم عامل ویندوز، ماژول‌ها با نام kernel-mode driver شناخته می‌شوند. ویندوز یک مدل خاص (windows driver model) طراحی کرده است که در آن یک چارچوب (kernel-mode driver framework) برای برنامه‌نویسی ماژول‌ها ارائه داده است. برای کدنویسی از دو زبان c و c++ می‌توان استفاده کرد و برای کامپایل آن باید از محیط نرم افزاری Microsoft Visual Studio استفاده کرد. دقت شود که کدنویسی برای

این سیستم عامل نیاز به درک عمیق از معماری آن دارد. همچنین یکی دیگر از تفاوت‌های آن با لینوکس این است که برای اضافه کردن این درایورها (ماژول)، نیاز دارد که توسط یک منبع معتبر به صورت دیجیتالی امضا شده باشند مگر اینکه سیستم عامل در حالت‌های خاصی مانند Test Mode راه‌اندازی شده باشند و به صورت پیش فرض، کاربر اجازه اضافه کردن هر ماژول دلخواهی به سیستم عامل خود را ندارد.

این فرایند به صورت مشابه در سیستم عامل macOS نیز دیده می‌شود، به این صورت که چارچوبی برای توسعه این ماژول‌ها وجود دارد و نیاز به امضا توسط منبع معتبر نیز وجود دارد.

بخش دوم: ساختمان داده‌های هسته

هسته لینوکس، به طور پیش فرض شامل یک سری ساختمان داده می‌باشد که می‌توانید برای توسعه کدهای خود از آن‌ها استفاده کنید.این ساختمان داده‌ها به نحوی تعریف شده‌اند که بتوان در کاربردهای مختلف از آن‌ها استفاده کرد و در عین حال، بهینه نیز باشند. بسیاری از این ساختمان داده‌ها به نحوی تعریف شده‌اند که برای استفاده از آن‌ها کافی است یک struct دلخواه تعریف کنید که یک عضو این استراکت، از یک نوع فیلد خاص از آن ساختمان داده باشد. برای درک بهتر این موضوع، به قطعه کد زیر توجه کنید:

```
// Define your data structure
struct myType
{
  struct rb_node nodeName; // needed to be capable to use with
                           // kernel data structure of red black trees
(rbtree.h)

  char *someData; // user defined fields
  int someOtherData;
  // ...
};
```

پس از تعریف struct دلخواهمان که فیلدهای مدنظرمان و فیلدهای مورد نیاز برای کار با کتاب‌خانه‌ها را دارد، می‌توانیم از توابع آماده ای که تعریف شده‌اند استفاده کنیم. به عنوان مثال ماکروی container_of یک ماکروی پر استفاده است که در آن با استفاده از آدرس (اشاره‌گر) یک فیلد در یک struct دلخواه، اشاره‌گر به ابتدای شیء از نوع struct دلخواه ما برگردانده می‌شود:

```
struct myType *data = container_of(node, struct myType, nodeName);
```

در کد بالا، فرض شده است ما آدرس فیلد nodeName را در متغیر node داریم، و struct مد نظر ما از نوع myType می‌باشد. پس از اجرای این ماکرو، متغیر data آدرس به داده از نوع myType را دارد.

بسیاری از ماکروها و توابع مورد استفاده در ساختمان داده‌های هسته، به همین صورت یا مشابه آن تعریف می‌شوند. در ادامه با یک نوع از این ساختمان داده‌ها بیشتر آشنا می‌شوید. مهم‌ترین ساختمان داده‌های هسته عبارت‌اند از:

- list_head:

ساختمان داده لیست پیوندی دوطرفه است که سرآیند آن در <linux/list.h> و <linux/types.h> می‌باشد. یکی از موارد استفاده این ساختمان داده در لینوکس برای مدیریت فرایندها (process) و device driver می‌باشد.

- rb_node, rb_root:

دو struct که برای پیاده‌سازی درخت قرمز-سیاه استفاده می‌شوند و سرآیند آن در<linux/rbtree.h> تعریف شده است. این ساختمان داده در لینوکس برای زمان بند و مدیریت حافظه استفاده شده است.

- hlist_head, hlist_node:

دو struct که برای ساخت hash table استفاده می‌شوند و سرآیند آن در<linux/hashtable.h> تعریف شده است. مدیریت اتصالات و مسیریابی در شبکه از موارد استفاده این ساختمان داده در لینوکس می‌باشند.

- Kfifo:

structیی است که برای ساخت یک صف FIFO استفاده می‌شود. سرآیند آن در <linux/kfifo.h> تعریف شده است. این ساختمان داده در ارتباط بین فرایندها در لینوکس استفاده شده است.

حال برای آشنایی بیشتر، یک نمونه کد با استفاده از list_head می‌نویسیم. برای ایجاد یک لیست پیوندی با استفاده از آن، باید یک struct شامل عناصر یک node تعریف کنید و یک فیلد دیگر از نوع list_head نیز در آن تعریف کنید:

```
struct birthday {
int day;
int month;
int year;
struct list_head list;
}
```

دقت شود که فیلد list_head دو عضو next و previous دارد که با ماکروهایی که در <linux/list.h> تعریف می‌شوند از این دو فیلد برای پیمایش روی لیست استفاده می‌کنند.

درج عناصر در لیست پیوندی

ماکروی LIST_HEAD برای ساخت یک لیست جدید با نامی‌که به عنوان ورودی به آن می‌دهیم استفاده می‌شود:

```
LIST_HEAD(my_list);
```

در ادامه برای ساخت عضوهای جدید در لیست، به صورت زیر عمل می‌کنیم:

```
struct birthday *b1;
b1 = kmalloc(sizeof(person), GFP_KERNEL);
b1->day = 2;
b1->month = 8;
b1->year = 1995;
INIT_LIST_HEAD(&b1->list);
```

تابع kmalloc() معادل هسته‌ای تابع سطح کاربری malloc() برای تخصیص حافظه است، جز اینکه در اینجا حافظه هسته تخصیص داده می‌شود (پرچم GFP_KERNEL برای تخصیص حافظه از حافظه اختصاص داده شده برای kernel و به صورت عادی می‌باشد. پرچم‌های دیگری نیز مانند GFP_ATOMIC, GFP_USER نیز وجود دارند). دقت داشته باشید برای استفاده از kmalloc() باید از کتاب‌خانه <linux/slab.h> استفاده کنید. ماکروی INIT_LIST_HEAD، عضو list در struct birthday را مقداردهی اولیه می‌کند. در واقع برای مقداردهی فیلد از نوع struct list_head در یک struct استفاده می‌شود. همچنین برای اضافه کردن اضافه کردن این عضو به انتهای لیست، باید از تابع list_add_tail استفاده شود:

```
list_add_tail(&b1->list, &my_list) // new node, head node
```

پیمایش لیست پیوندی

برای پیمایش روی یک لیست پیوندی روش‌های مختلفی وجود دارد . معروف‌ترین آن ماکروی list_for_each_entry می‌باشد. این ماکرو سه آرگومان ورودی می‌گیرد:

- اشاره‌گری به رکوردی که پیمایش روی آن صورت می‌گیرد (یعنی در هنگام پیمایش، اشاره‌گر به عضو فعلی در این متغیر ذخیره می‌شود).
- اشاره‌گر به ابتدای لیست
- نام متغیر از نوع struct list_head در استراکت مد نظر

کد زیر این ماکرو را به تصویر می‌کشد:

```
Struct birthday *current;
list_for_each_entry(current, &my_list, list) {
  // some computation with the current node
}
```

تمرین‌ها: (خروجی مورد انتظار آزمایش)

تمرین ۱:

برای آشنایی بیشتر با ماژول‌ها و نحوه استفاده از توابع هسته، یکی از موارد زیر را به عنوان تمرین انتخاب کرده و کد آن را به مسئول آزمایشگاه تحویل دهید.

الف) یک ماژول بنویسید که بتوانید به آن پارامترهایی از نوع short, int, long, string, array به عنوان ورودی بدهید و از آن در کد استفاده کنید (راهنمایی: می‌توانید درباره module_param و MODULE_PARM_DES جستجو کنید).

ب) یک ماژول بنویسید که در آن بتوانید اطلاعات فرایندها (process) را از هسته دریافت کنید و مواردی مانند شناسه فرایند، میزان مصرف CPU، و میزان مصرف حافظه (RAM) آن را چاپ کنید (راهنمایی: درباره struct task_struct و for_each_process جستجو کنید).

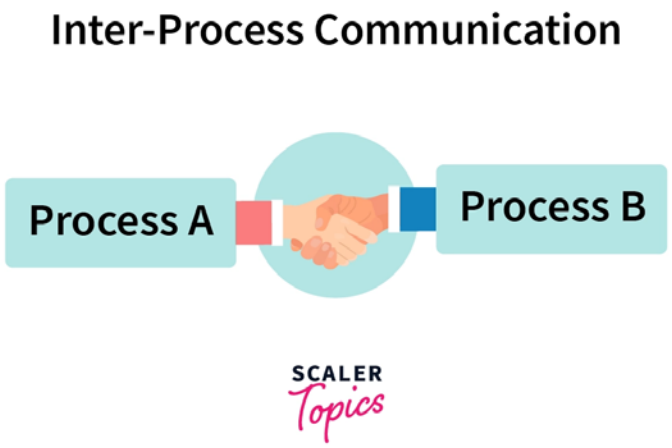
تمرین ۲:

یک ماژول بنویسید که در تابع ابتدایی آن، پنج عنصر struct birthday ایجاد کند و آن را پیمایش کرده و اطلاعات هر عضو را چاپ کند. همچنین در هنگام خروج نیز این لیست را به صورت برعکس (reverse) پیمایش کنید و سپس هر کدام را از لیست حذف کرده و فضای آزاد شده را به هسته برگردانید.

تمرین امتیازی:

یک مثال ساده دیگر با استفاده از rbtree یا hashtable را انجام دهید.

آزمایش چهارم: ارتباط بین پردازه‌ها



هدف آزمایش

آشنا شدن با نحوه‌ی برقراری ارتباط بین دو پردازه
توسط دو روش:
۱) حافظه مشترک
Shared Memory
و
۲) ارسال پیام
Message Passing

آمادگی پیش از آزمایش:

–

شرح آزمایش

مقدمه

فرآیندها در سیستم عامل دو نوع هستند:

- فرآیندهای مستقل
- فرآیندهای همکار

فرآیندهای مستقل آن دسته از فرآیندهایی هستند که نه تحت تأثیر سایر فرآیندهای در حال اجرا در سیستم عامل قرار می‌گیرند و نه بر سایر فرایندها تاثیر می‌گذارند. به عبارت دیگر، هر فرآیندی که داده‌ها را با سایر فرآیندها به اشتراک نمی‌گذارد، یک فرآیند مستقل است.

فرآیندهای همکار به آن دسته از فرآیندهایی گفته می‌شود که می‌توانند بر سایر فرآیندهای در حال اجرا در سیستم تاثیر بگذارند یا تحت تاثیر قرار گیرند. به عبارت دیگر، فرآیندی که داده‌ها را با سایر فرآیندها به اشتراک می‌گذارد، یک فرآیند همکار است.

ارتباطات بین فرایندی برای ایجاد فرآیندهای همکار مفید است و به سازوکارها و روش‌هایی اشاره دارد که توسط فرآیندها برای برقراری ارتباط و همگام سازی با یکدیگر استفاده می‌شود. IPC به فرآیندها اجازه می‌دهد تا به وظایف تبادل داده‌ها و اشتراک‌گذاری اطلاعات، بالا بردن سرعت محاسبات (برای سریع انجام شدن کاری آن را به فرایندهای کوچکتر تقسیم و به صورت موازی اجرا می‌کنیم) و پیمانه‌ای بودن (کارهای مختلف یک سیستم را بین چند فرایند تقسیم می‌کنیم) بپردازند.

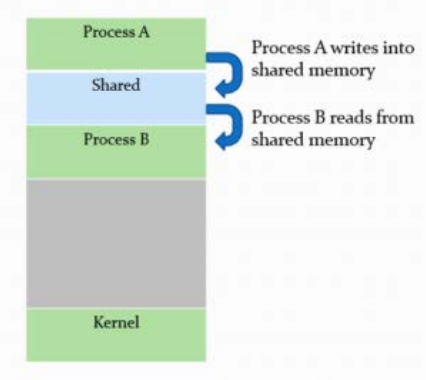
به طور کلی دو رویکرد برای اجرای ارتباطات بین فرایندی وجود دارد:

۱- حافظه مشترک (shared memory)

۲- ارسال پیام (message passing)

حافظه مشترک

فرآیندهای همکار یک قسمت از حافظه را به اشتراک می‌گذارند. این سریعترین روش برای ارتباطات بین فرایندی است. سیستم عامل یک بخش حافظه مشترک در RAM ایجاد می‌کند تا چندین فرآیند بتوانند در آن بخش حافظه بخوانند و بنویسند. فرآیندها بدون فراخوانی توابع سیستم عامل، این قطعه حافظه را به اشتراک می‌گذارند.

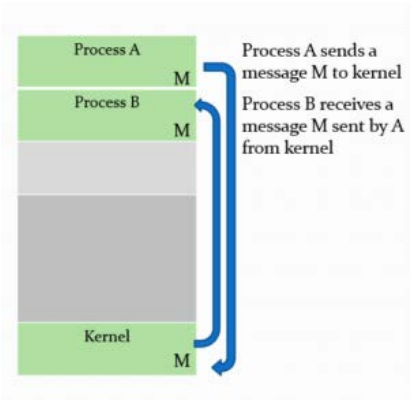


ارسال پیام

این روش دوم برای ارتباطات بین فرایندی است. دو عملیات برای ارتباط فرآیندها فراهم می‌کند:

- فرستادن پیام
- دریافت پیام

این روش به این دلیل که از system call برای برقراری ارتباط بین فرآیندها استفاده می‌کند، در مقایسه با روش حافظه مشترک کندتر است.



توابع C برای حافظه مشترک

کتاب‌خانه مورد استفاده: sys/shm.h

1. shmget: این تابع یک بخش حافظه مشترک جدید ایجاد می‌کند یا اگر وجود داشته باشد شناسه یک بخش موجود را بر می‌گرداند.

```
int shmget(key_t key،size_t size،int shmflg);
```

- مقدار پارامتر key را IPC_PRIVATE قرار می‌دهیم که ایجاد یک قطعه حافظه مشترک جدید را درخواست می‌کند که تنها با شناسه قطعه که خروجی این تابع است قابل دسترسی است.
- پارامتر size ، اندازه مورد نیاز قطعه حافظه را مشخص می‌کند. اگر قطعه از قبل وجود داشته باشد، اندازه نباید از اندازه مشخص شده در زمان ایجاد قطعه بیشتر باشد.
- پارامتر flags صفر یا چند گزینه را مشخص می‌کند. برای مشخص نکردن هیچ flag ای آن را ۰ می‌دهیم و برای چند مقدار از عملگر | استفاده می‌شود. برای مثال با قرار دادن مقدار S_IWUSR | S_IRUSR دسترسی خواندن و نوشتن می‌دهیم.

این دستور در صورت موفقیت یک مقدار integer بر می‌گرداند که به عنوان identifier برای این حافظه اشتراکی است و سایر فرآیندها برای استفاده از حافظه مشترک باید این identifier را مشخص کنند .

2. shmat: این تابع یک بخش حافظه مشترک را به فضای آدرس فرایندی که آن را فرا می‌خواند اضافه می‌کند و به آن امکان دسترسی به حافظه مشترک را می‌دهد.

```
void *shmat(int id, const void *addr, int flags);
```

- پارامتر id، id یک قطعه حافظه مشترک است (همانی که در خروجی shmget هم دیدیم).
- پارامتر addr یک مقدار pointer را مشخص می‌کند که نشان دهنده آدرسی است که قطعه حافظه مشترک باید به آن متصل شود. برای این آزمایش آن را NULL میگذاریم تا این آدرس توسط سیستم انتخاب شود.
- پارامتر flags می‌تواند دسترسی‌ها به این قطعه را مشخص کند. برای این آزمایش آن را ۰ می‌گذاریم.

3. shmdt: این تابع یک قطعه حافظه مشترک را از فضای آدرس فرایندی که این تابع را فراخوانی کرده جدا می‌کند.

```
int shmdt(const void *addr);
```

- پارامتر addr یک مقدار pointer را مشخص می‌کند که نشان دهنده آدرسی است که قطعه حافظه مشترک باید به آن متصل شود.

4. shmctl: این تابع عملیات کنترلی را در بخش حافظه مشترک انجام می‌دهد، مانند حذف یا تغییر مجوزهای آن.

```
int shmctl(int id, int cmd, struct shmid_ds *buf);
```

- پارامتر id، id یک قطعه حافظه مشترک است.
- پارامتر cmd عملیات خاصی که باید توسط shmctl انجام شود را مشخص می‌کند (مقدار IPC_RMID نیازی به مشخص کردن پارمتر buf ندارد و حافظه مشترک و id آن را از سیستم حذف می‌کند. مقدار IPC_SET برای تغییر مالکیت یا قوانین دسترسی قطعه حافظه مشترک است. مقدار IPC_STAT محتویات shared memory id data structure را برمی‌گرداند و آن را در buf ذخیره می‌کند).

تمرین‌ها: (خروجی مورد انتظار آزمایش)

تمرین ۱:

شرایطی را تصور کنید که در آن شما دو فرآیند دارید، یک فرآیند تولید کننده و یک فرآیند مصرف کننده، که از طریق حافظه مشترک با هم ارتباط برقرار می‌کنند. فرآیند تولید کننده اعداد تصادفی را تولید می‌کند و آنها را در حافظه مشترک ذخیره می‌کند، در حالی که فرآیند مصرف کننده این اعداد را می‌خواند و مجموع آنها را محاسبه می‌کند.

تمرین ۲:

حالت پایه

در این آزمایش باید یک برنامه کاربردی مدیریت انبارگردانی (Warehouse management) را به وسیله‌ی زبان C پیاده‌سازی کنید. این برنامه دو بخش دارد: سرور و کاربر. سرور اطلاعات مربوط به اجناس انبار را نگه‌داری می‌کند و وقتی از سمت انبارگردان ،به عنوان کاربر، پیامی‌دریافت می‌کند بسته به نوع پیام عملیات مدنظر کاربر را انجام می‌دهد. اطلاعات اجناس یک انبار شامل یک لیست از اسامی‌اجناس موجود و مقدار موجودی هر یک می‌باشد.

راه‌اندازی سرور و کاربر توسط دستورات زیر انجام می‌شود:

```
server [server-port-number]
client [server-host-name] [server-port-number] [client-name]
```

وقتی کاربر به سرور متصل شد، دستورات زیر باید پشتیبانی شوند:

list:
لیست تمامی‌کالاهای موجود انبار به همراه مقدار موجودی هر یک را به کاربر نشان می‌دهد.
create [new_product_name]:
کالای جدید با نام مشخص شده و با مقدار اولیه صفر به لیست کالاهای انبار اضافه می‌شود.
create [new_product_name] [initial_count]:
کالای جدید با نام و با مقدار اولیه مشخص شده به لیست کالاهای انبار اضافه می‌شود.
add [product_name] [amount]:
به مقدار موجودی کالای مشخص شده به اندازه تعیین شده اضافه شود.
reduce [product_name] [amount]:
از مقدار موجودی کالای مشخص شده به اندازه تعیین شده کم شود.
remove [product_name]:
کالای مشخص شده از لیست انبار حذف شود.
quit:
کاربر کلا از برنامه خارج می‌شود.

دقت شود در پیاده‌سازی موارد زیر حتما رعایت شوند:

- در حالت پایه آزمایش در هر لحظه تنها یک کاربر به سرور متصل است.
- مقدار موجودی یک کالا در انبار هرگز نمی‌تواند کمتر از صفر باشد (حتی موقع create).
- از ایجاد کالاهایی با اسم تکراری جلوگیری شود.
- برای ایجاد تغییر در مقدار یک کالا یا حذف آن، نام آن حتما در لیست اجناس وجود داشته باشد.
- برای آنکه یک کالا remove شود، بایستی حتما موجودی آن برابر صفر باشد.
- در همه مواردی که خطا رخ می‌دهد، پیغام مناسب داده شود.

حالت پیشرفته

در این حالت همه موارد ذکر شده در حالت پایه باید پیاده‌سازی شوند، فقط با این تفاوت که دیگر محدود به یک کاربر نیستیم. چندین کاربر همزمان می‌توانند به سرور متصل شوند و سرور باید بتواند برای هر یک از کاربران بطور جداگانه لیست اجناس را نگه‌داری کند. همچنین کاربران در این حالت باید قادر باشند تا از اجناس موجود خود تعدادی را به کاربری دیگر انتقال دهند. برای پشتیبانی از این عملیات دستور زیر نیز اضافه بر دستورات حالت پایه باید پشتیبانی شود:

send [target_client_name] [product_name] [amount]:
در این دستور، یک کاربر ۳ مورد را تعیین می‌کند، به کدام کاربر قرار است جنس انتقال داده شود (target_client_name)، چه جنسی قرار است انتقال داده شود (product_name) و چه مقدار جنس قرار است انتقال پیدا کند (amount).

دقت شود در پیاده‌سازی موارد زیر حتما رعایت شوند:

- کاربرانی با client_name تکراری نداشته باشیم.
- جنسی که برای انتقال تعیین شده در لیست کاربر مبدأ وجود داشته باشد و موجودی کافی داشته باشد.
- اگر جنسی که انتقال می‌یابد در لیست اجناس کاربر مقصد (target_client) وجود ندارد، به لیست اجناس او به عنوان کالای جدید با مقدار مشخص شده اضافه شود.
- انتقال با amount منفی نداریم.
- برای مثالی از ارتباط client و server به نمونه کد زیر توجه کنید.

کد مربوط به client:

```c
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>

#define PORT 8080
int main(int argc, char const *argv[]) {
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char *hello = "Hello from client";
    char buffer[1024] = {0};
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }
    // sets all memory cells to zero
    memset(&serv_addr, '0', sizeof(serv_addr));

    // sets port and address family
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0)
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }

    // connects to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
```

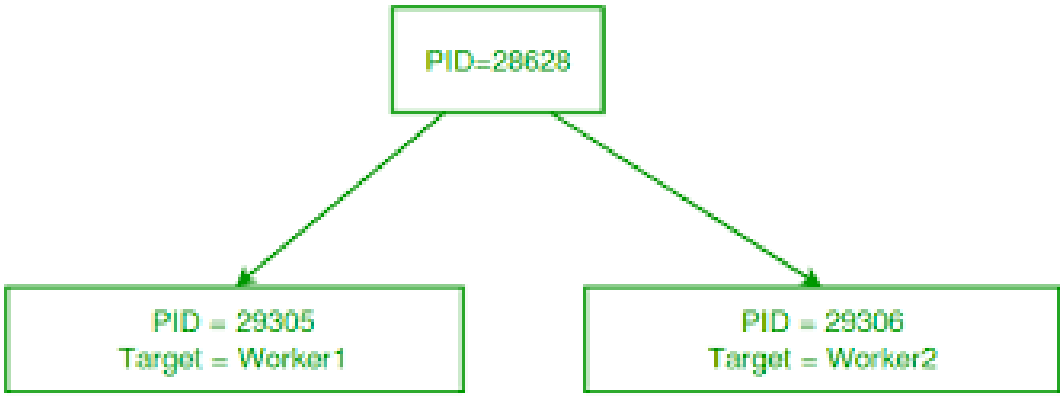
```
    {
        printf("\nConnection Failed \n");
        return -1;
    }
    send(sock, hello, strlen(hello), 0);
    printf("Hello message sent\n");
    valread = read(sock, buffer, 1024);
    if (valread < 0) {
    perror("read");
    return -1;
    }
    printf("%s\n", buffer);
    return 0;
}
```

کد مربوط به server:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>

#define PORT 8080

int main(int argc, char const *argv[]) {
    char buffer[1024] = {0};
    char *hello = "Hello from server";

    // creates socket file descriptor
    int server_fd;
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    struct sockaddr_in address;
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT); // host to network -- coverts the ending
of the given integer
    const int addrlen = sizeof(address);
```

```
    // binding
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    // listening on server socket with backlog size 3.
    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    printf("Listen on %s:%d\n", inet_ntoa(address.sin_addr),
ntohs(address.sin_port));

    // accepting client
    // accept returns client socket and fills given address and addrlen with
client address information.
    int client_socket, valread;
    if ((client_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen)) < 0) {
        perror("accept");
        exit(EXIT_FAILURE);
    }
    printf("Hello client %s:%d\n", inet_ntoa(address.sin_addr),
ntohs(address.sin_port));

    // reads a buffer with maximum size 1024 from socket.
    valread = read(client_socket, buffer, 1024);
    if (valread < 0) {
        perror("read");
        exit(EXIT_FAILURE);
    }
    printf("(s = %d) %s\n", valread, buffer);

    // writes to client socket
    send(client_socket, hello, strlen(hello), 0);
    printf("Hello message sent\n");
    return 0;
}
```

آزمایش پنجم:
برنامه‌نویسی چند فرآیندی



هدف آزمایش

برنامه‌نویسی چند فرآیندی و رسم نمودار
توزیع نرمال

آمادگی پیش از آزمایش:

-

شرح آزمایش

مقدمه:

در روش‌های تحقیقات علمی و آمار ، بررسی نمونه‌ای (Sampling) به فرآیندی گفته می‌شود که براساس آن انتخاب اعضایی از جامعه آماری صورت می‌پذیرد. این کار با هدف ایجاد برآورد از جامعه آماری و یا شناخت بیشتر آن انجام می‌شود. اهمیت نمونه‌گیری را می‌توان صرفه‌جویی در زمان برای تهیه مشاهدات از جامعه آماری به منظور انجام تحقیق علمی‌دانست. معمولا نمونه‌گیری در مقابل سرشماری قرار دارد. سرشماری به منظور بررسی همه اعضای جامعه آماری به کار می‌رود ولی گاهی دسترسی به تمام اعضای این جامعه میسر نیست یا تعداد اعضای آن نامتناهی است.

یک روش رایج برای برخی محاسبات در ریاضی روش نمونه برداری است. برای مثال می‌توان عدد π را با همین روش محاسبه کرد. در این فرایند با استفاده از تولید زوج عددهای تصادفی فراوان (به تعداد نمونه‌ها) و تشخیص اینکه هر زوج در مساحت دایره قرار می‌گیرد یا خیر و تقسیم آن‌ها به یکدیگر، عدد π محاسبه می‌شود. برای درک بهتر می‌توانید از مرجع (۴) در این آزمایش استفاده کنید.

تعریف مسئله:

در این آزمایش هدف آن است که با استفاده از نمونه برداری، نمودار توزیع نرمال ترسیم شود. در ابتدا یک آرایه با نام hist که ۲۵ خانه دارد بسازید. از این آرایه برای نگهداری نتایج آزمایش استفاده می‌شود. این ۲۵ خانه نمایندگان اعداد -۱۲ و +۱۲ هستند. فرآیند نمونه برداری به این صورت است که مقدار ابتدایی متغیر counter شما با مقدار صفر شروع می‌شود و شما بایستی در ۱۲ مرحله و در هر مرحله یک عدد تصادفی بین ۰ تا ۱۰۰ تولید کنید. اگر این عدد تصادفی بزرگتر یا مساوی ۴۹ بود، مقدار counter را یکی افزایش دهید و برعکس. پس از پایان ۱۲ مرحله، بر اساس مقدار counter خانه مربوطه از آرایه hist را افزایش دهید.

شرح کلی:

گام‌های زیر را انجام دهید:

۱. ابتدا کد برنامه ای که در تعریف مسئله شرح داده شد را در حالت سریال بنویسید و زمان اجرا شدن برنامه خود را در جدول زیر گزارش دهید.

تعداد نمونه	۵۰۰۰	۵۰۰۰۰	۵۰۰۰۰۰
زمان اجرا			

۲. حال برنامه ای بنویسید که با استفاده از fork() و یا exec() تعدادی فرآیند فرزند ایجاد شود و کارها را پخش کنید.
قطعه کد زیر مثالی از نحوه استفاده از fork() است. خروجی این کد در زیر آن آمده است.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void forkexample(){
   int x = 1;
   if (fork() == 0)
       printf("Child has x = %d\n",++x);
   else
       printf("Parent has x = %d\n", --x);
}
int main(){
   forkexample();
   return 0;
}
```

خروجی:

```
Parent has x = 0
Child has x = 2
(or)
Child has x = 2
Parent has x = 0
```

نکته) از مطالبی که در جلسه قبل (IPC) آموخته اید برای ارتباط بین فرآیندهای فرزند و والد استفاده کنید.

زمان اجرای خود را در جدول زیر گزارش دهید.

تعداد نمونه	۵۰۰۰	۵۰۰۰۰	۵۰۰۰۰۰
زمان اجرا			

۳. آیا این برنامه درگیر شرایط مسابقه می‌شود؟ چگونه؟ اگر جوابتان مثبت بود راه حلی برای آن بیابید.

۴. نتایج قسمت اول و دوم را مقایسه کنید و میزان افزایش سرعت را در جدول گزارش دهید.

تعداد نمونه	۵۰۰۰	۵۰۰۰۰	۵۰۰۰۰۰
زمان اجرا			

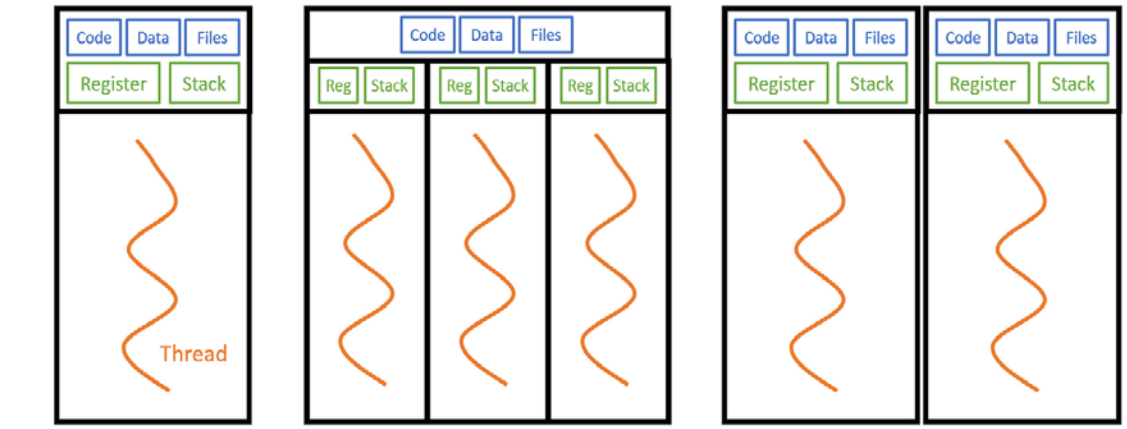
با استفاده از قطعه کد زیر می‌توانید نتایج حاصل از محاسبات را ترسیم کنید.

```c
void printHistogram(int* hist){
   int i, j;
   for(i = 0; i < 25; i++){
       for(j = 0; j < hist[i]; j++){
           printf("*");
       }
       printf("\n");
   }
}
```

مراجع مطالعه/پیوست‌ها:

1. https://www.geeksforgeeks.org/fork-system-call/
2. https://www.geeksforgeeks.org/exec-family-of-functions-in-c/
3. http://www.cls.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
4. https://www.youtube.com/watch?v=VJTFfIqO4TU

آزمایش ششم:
برنامه‌نویسی چند نخی



هدف آزمایش

برنامه‌نویسی چند نخی و محاسبه‌ی
Prefix Sum

آمادگی پیش از آزمایش:

-

شرح آزمایش

مقدمه:

از برنامه‌نویسی چند فرآیندی و چند نخی، می‌توان برای افزایش سرعت اجرای برنامه استفاده کرد، که در بخش فرآیندها از برنامه‌نویسی چند فرآیندی برای افزایش سرعت اجرای برنامه استفاده کردید. گاهی اوقات مسئله در نگاه اول به نظر می‌رسد قابل موازی سازی نیست، اما با پیاده‌سازی دوباره الگوریتم با دید موازی سازی، می‌توان آن را موازی کرد و در زمان کمتر از حالت سریال اجرا کرد.

تعریف مسئله:

مسئله Prefix Sum:

فرض کنید یک آرایه a داریم که در هر خانه اش عددی نوشته شده است. حال می‌خواهیم آرایه b بسازیم به طوری که:

b[i] = sum(a[0..i])

برای مثال اگر آرایه a برابر آرایه زیر باشد:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
۱	۲	۱	۵	۰	۷	۳	۴	۲	۷

آرایه b ای که از a ساخته می‌شود برابر است با:

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]
۱	۳	۴	۹	۹	۱۶	۱۹	۲۳	۲۵	۳۲

۱. یک برنامه سریال برای محاسبه این کار با پیچیدگی زمانی O(n) بنویسید.

۲. آیا می‌توانید این برنامه را موازی کنید؟ مشکل چیست؟

۳. آرایه a را به دو تکه تقسیم کنید، prefix sum هر دو تکه را به طور موازی محاسبه کنید، تکه ابتدایی b با روش سریال آیا تفاوتی دارد؟ تکه پایانی چطور؟ چه عددی باید با تمامی‌خانه‌های تکه پایانی جمع شود؟

۴. این عدد را به طور موازی با خانه‌های تکه پایانی جمع کنید. برای این کار نیز می‌توانید به طور موازی عمل کنید.

۵. خانه‌های a را با اعداد دلخواه پر کنید (مراقب باشید برای محاسبه درایه‌های آرایه b، سرریز رخ ندهد)، حال برای تعداد مشخص شده در جدول زیر، زمان اجرای الگوریتم‌ها را مقایسه کنید.

تعداد نمونه	۵۰۰۰	۵۰۰۰۰	۵۰۰۰۰۰
زمان اجرای برنامه سریال			
زمان اجرای برنامه دو نخه			

راهنمایی: برای مطالعه نحوه استفاده از نخ در زبان C، لینک مرجع قرار داده شده را مطالعه فرمایید.

یک نمونه از برنامه چند نخه در زیر نمایش داده شده است:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>
 // A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
   sleep(1);
   printf("Printing GeeksQuiz from Thread \n");
   return NULL;
}

int main()
{
   pthread_t thread_id;
   printf("Before Thread\n");
   pthread_create(&thread_id, NULL, myThreadFun, NULL);
   pthread_join(thread_id, NULL);
   printf("After Thread\n");
   exit(0);
}
```

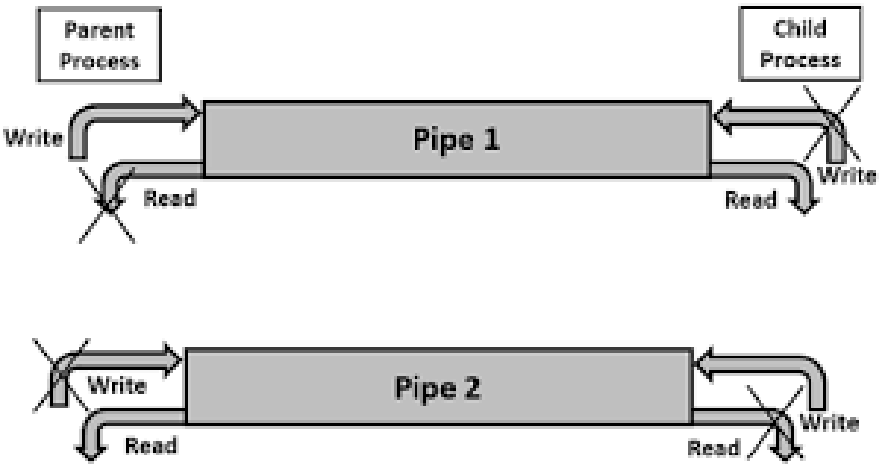
خروجی‌های مورد انتظار آزمایش:

- انتظار می‌رود بخش تمرین‌ها به‌صورت کامل توسط دانشجویان انجام شود و نتیجه به مدرس آزمایشگاه تحویل داده شود.
- تمامی مراحل انجام تمرین باید مرحله به مرحله توسط تمامی دانشجویان انجام شده و تمامی‌آن‌ها باید پس از انجام آزمایش قادر به توضیح مراحل مختلف باشند.

مراجع مطالعه/پیوست‌ها:

https://www.geeksforgeeks.org/multithreading-in-c/

آزمایش هفتم:
لوله (Pipe)



هدف آزمایش

آشنایی با فراخوانی سیستمی لوله

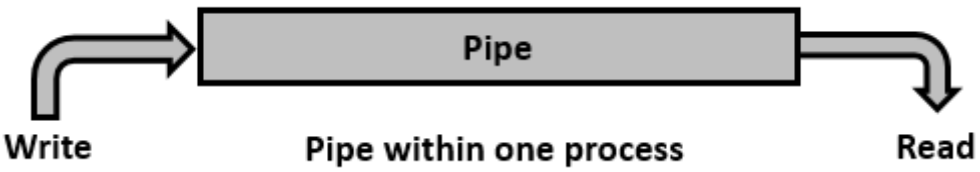
آمادگی پیش از آزمایش:

-

شرح آزمایش

مقدمه:

Pipe یا لوله یک واسطه ارتباطی بین دو یا چند فرایند است. این ارتباط با نوشتن یک فرایند در pipe و خواندن فرایند دیگر از آن برقرار می‌شود. برای اجرای فراخوانی سیستم لوله (pipe system call)، دو فایل ایجاد می‌شود، یکی برای نوشتن در فایل و دیگری برای خواندن از آن. مکانیزم pipe را می‌توان در یک سناریوی واقعی مشاهده کنیم، مانند پر کردن آب با لوله در یک سطل، و برداشتن از آن، مثلاً با یک لیوان. فرایند پر کردن همان نوشتن در لوله است و فرایند خواندن همان برداشتن یا خواندن از لوله است. این بدان معناست که یک خروجی (آب) ورودی برای دیگری (سطل) است.



برای آزمایش این بخش از توابعی که در ادامه معرفی می‌شود استفاده می‌کنیم.

```
#include<unistd.h>
int pipe(int pipefd[2]);
```

تابع pipe() یک خط لوله یک طرفه ایجاد می‌کند که می‌تواند برای ارتباطات بین فرایندی (Inter-process communication) استفاده شود. پس از اجرا شدن، این تابع دو توصیفگر فایل (file descriptor) می‌سازد، pipefd[0] و pipefd[1].

pipefd[0] سمت خواندن لوله است (reading end)، بنابراین پردازه‌ای که قرار است داده را دریافت کند باید از این file descriptor استفاده کند.

pipefd[1] سمت نوشتن لوله (writing end) است، بنابراین پردازه‌ای که قرار است داده را ارسال کند بایستی از این file descriptor استفاده کند. این فراخوانی در صورت موفقیت صفر و در صورت شکست -۱ برمی‌گرداند.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

با وجود اینکه خواندن و نوشتن عملیات اصلی فایل هستند، باز کردن فایل قبل از انجام عملیات و بستن فایل پس از اتمام عملیات ضروری است. معمولاً به‌طور پیش‌فرض برای هر فرآیند ۳ توصیف‌گر فایل باز می‌شود:

ورودی (standard input – stdin)،

خروجی (standard output – stdout)

و خطا (standard error – stderr)

که به‌ترتیب با داشتن توصیف‌گر فایل ۰، ۱ و ۲ استفاده می‌شود. این فراخوانی سیستم یک توصیفگر فایل را برمی‌گرداند که برای عملیات بیشتر فایل (خواندن، نوشتن و جستجو) استفاده می‌شود. معمولاً توصیفگرهای فایل از ۳ شروع می‌شود و با باز شدن تعداد فایل‌ها افزایش می‌یابد.

آرگومان‌های ارسال شده به فراخوانی سیستم open عبارتند از pathname و flageهایی که هدف باز کردن فایل را ذکر می‌کنند (مثلاً برای خواندن، O_RDONLY، برای نوشتن، O_WRONLY، برای خواندن و نوشتن، O_RDWR، برای اضافه کردن به فایل موجود O_APPEND، برای ایجاد فایل، در صورت عدم وجود با O_CREAT و غیره) و حالت(mode) مورد نیاز که مجوزهای خواندن، نوشتن و اجرا برای کاربر یا مالک، گروه یا دیگران را فراهم می‌کند. حالت (mode) را می‌توان با نمادها ذکر کرد: خواندن – ۴، نوشتن – ۲ و اجرا – ۱.

یادآوری: مقدار اکتال (با ۰ شروع می‌شود) ۰۷۶۴ نشان می‌دهد که مالک مجوزهای خواندن، نوشتن و اجرا را دارد، گروه مجوز خواندن و نوشتن دارد و دیگران مجوز خواندن دارند. این نیز می‌تواند به صورت S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH نشان داده شود.

این فراخوانی سیستمی، در صورت موفقیت، شناسه توصیفگر فایل جدید و در صورت بروز خطا -۱ را برمی‌گرداند. علت خطا را می‌توان با متغیر errno یا تابع perror() مشخص کرد.

```
#include<unistd.h>

int close(int fd)
```

فراخوانی سیستم بالا برای بستن توصیفگر فایلی که از قبل باز شده است می‌باشد. این بدان معناست که فایل، دیگر در حال استفاده نیست و منابعی که گرفته بود، می‌توانند توسط هر فرآیند دیگری دوباره استفاده شوند. این فراخوانی سیستم،

صفر را در صورت موفقیت و -۱ را در صورت خطا برمی‌گرداند. علت خطا را می‌توان با متغیر errno یا تابع perror() شناسایی کرد.

```
#include<unistd.h>

ssize_t read(int fd, void *buf, size_t count)
```

فراخوانی سیستم فوق برای خواندن از فایل مشخص شده با آرگومان‌های توصیفگر فایل fd، بافر مناسب با حافظه اختصاص داده شده (اعم از ایستا یا پویا) و اندازه بافر است. شناسه توصیفگر فایل برای شناسایی فایل مربوطه است که پس از فراخوانی سیستم open() یا pipe() برگردانده می‌شود. قبل از خواندن از روی فایل، فایل باید باز شود که در صورت فراخوانی pipe() به طور خودکار باز می‌شود.

این فراخوانی در صورت موفقیت تعداد بایت‌های خوانده شده (یا صفر در صورت مواجهه با انتهای فایل) را برمی‌گرداند و در صورت شکست -۱ را بر می‌گرداند. علت خطا و یا کمتر بودن بایت‌های خوانده‌شده نسبت به تعداد درخواست‌شده را می‌توان با متغیر errno یا تابع perror() شناسایی کرد.

```
#include<unistd.h>

ssize_t write(int fd, void *buf, size_t count)
```

فراخوانی سیستم فوق برای نوشتن روی فایل مشخص شده با آرگومان‌های توصیفگر فایل fd، یک بافر مناسب با حافظه اختصاص داده شده (اعم از ایستا یا پویا) و اندازه بافر است. شناسه توصیفگر فایل برای شناسایی فایل مربوطه است که پس از فراخوانی سیستم open() یا pipe() برگردانده می‌شود. قبل از نوشتن روی فایل، فایل باید باز شود که در صورت فراخوانی pipe() به طور خودکار باز می‌شود. این فراخوانی تعداد بایت‌های نوشته شده (یا صفر در صورتی که چیزی نوشته نشده باشد) و در صورت شکست (منفی یک) را برمی‌گرداند. علت خطا را می‌توان با متغیر errno یا تابع perror() شناسایی کرد.

برای مثال برنامه زیر را در نظر بگیرید که در آن پردازه والد یک پردازه فرزند ایجاد می‌کند؛ در ادامه پردازه والد یک لوله می‌سازد که از طریق سمت نوشتنش به فرزندش داده می‌فرستد، از طرفی فرزند هم داده را دریافت می‌کند و آن را روی صفحه از طریق سمت خواندن لوله چاپ می‌کند.

```
//Program to send a message from parent process to child process using pipe()
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
    int fd[2],n;
    char buffer[100];
    pid_t p;
    pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]
    p=fork();
```

```
    if(p>0) //parent
    {
        printf("Parent Passing value to child\n");
        write(fd[1],"hello\n",6); //fd[1] is the write end of the pipe
        wait();
    }
    else // child
    {
        printf("Child printing received value\n");
        n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe
        write(1,buffer,n);
    }
}
```

شرح آزمایش:

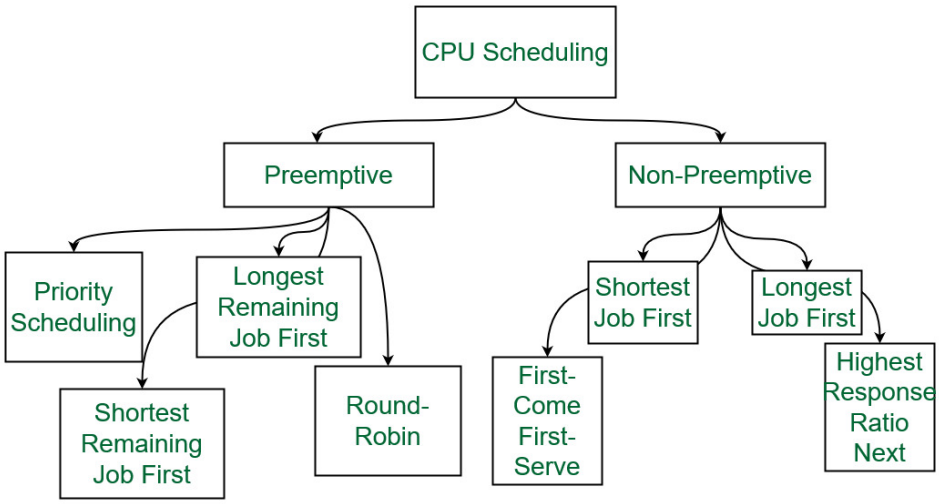
حال محیطی فراهم کنید که در آن دو فرآیند با استفاده از خط لوله به تبادل یک پیام متنی بپردازند. فرآیند اول یک پیام متنی دارای حروف بزرگ و کوچک (برای مثال: This Is First Process) به فرآیند دوم ارسال می‌کند، فرآیند دوم این پیام را دریافت می‌کند و حروف بزرگ را به حروف کوچک و حروف کوچک را به حروف بزرگ تبدیل می‌کند (برای مثال tHIS iS fIRST pROCESS) و به فرآیند اول می‌فرستد.

راهنمایی: برای این کار به دو خط لوله نیاز دارید.

خروجی مورد انتظار آزمایش:

- انتظار می رود دانشجویان آزمایش را به طور کامل انجام دهند و برای توضیح کامل آن به مسئول آزمایشگاه آماده باشند.

آزمایش هشتم:
زمانبندی CPU



هدف آزمایش

پیاده‌سازی الگوریتم‌های زمان‌بندی

آمادگی پیش از آزمایش:

مطالعه الگوریتم‌های زمان‌بندی

شرح آزمایش

بخش اول: پیاده‌سازی الگوریتم‌های زمان‌بندی تدریس شده

استاد آزمایشگاه یکی از الگوریتم‌های زمان‌بندی زیر را انتخاب نمایند و دانشجویان آن الگوریتم را به شیوه توضیح داده شده در زبان C پیاده‌سازی کنند:

الگوریتم زمان‌بندی FCFS:

برنامه‌ای بنویسید که الگوریتم زمان‌بندی First Come First Serve را پیاده‌سازی کند، برای این کار مراحل زیر را انجام دهید:

۱. تعداد فرایندها را از کاربر دریافت کنید.
۲. زمان سرویس‌دهی هر فرایند را از کاربر دریافت کنید.
۳. زمان انتظار برای فرایند اول را با صفر مقدار دهی کنید.
٤. زمان انتظار سایر فرایندها را مشخص کنید (دقت کنید که زمان انتظار یک فرایند برابر با زمان اجرای فرایند قبل است، زمان اجرای یک فرایند برابر است با مجموع زمان انتظار و زمان سرویس‌دهی).
٥. زمان اجرای فرایندها را حساب کنید.
٦. متوسط زمان انتظار و زمان اجرا برای هر فرایند را حساب کنید و نمایش دهید.

الگوریتم زمان‌بندی SJF:

برنامه‌ای بنویسید که الگوریتم زمان‌بندی Shortest Job First را پیاده‌سازی کند، برای این کار مراحل زیر را انجام دهید:

۱. تعداد فرایندها را از کاربر دریافت کنید.
۲. زمان سرویس‌دهی هر فرایند را از کاربر دریافت کنید.
۳. زمان انتظار برای فرایند اول را با صفر مقدار دهی کنید.
٤. فرایندها را را بر اساس زمان سرویس‌دهی مرتب کنید.
٥. زمان انتظار سایر فرایندها را مشخص کنید (دقت کنید که زمان انتظار یک فرایند برابر با زمان اجرای فرایند قبل است، زمان اجرای یک فرایند برابر است با مجموع زمان انتظار و زمان سرویس‌دهی).
٦. زمان اجرای فرایندها را حساب کنید.

۷. متوسط زمان انتظار و زمان اجرا برای هر فرایند را حساب کنید و نمایش دهید.

الگوریتم زمان‌بندی Priority:

برنامه‌ای بنویسید که الگوریتم زمان‌بندی اولویت دار (priority) را پیاده‌سازی کند. برای این کار، مراحل زیر را انجام دهید:

۱. تعداد فرایندها را از کاربر دریافت کنید.
۲. زمان سرویس‌دهی و درجه اهمیت هر فرایند را از کاربر دریافت کنید.
۳. زمان انتظار برای فرایند اول را با صفر مقدار دهی کنید.
٤. فرایندها را را بر اساس درجه اهمیت مرتب کنید.
٥. زمان انتظار سایر فرایندها را مشخص کنید (دقت کنید که زمان انتظار یک فرایند برابر با زمان اجرای فرایند قبل است، زمان اجرای یک فرایند برابر است با مجموع زمان انتظار و زمان سرویس‌دهی).
٦. زمان اجرای فرایندها را حساب کنید.
۷. متوسط زمان انتظار و زمان اجرا برای هر فرایند را حساب کنید و نمایش دهید.

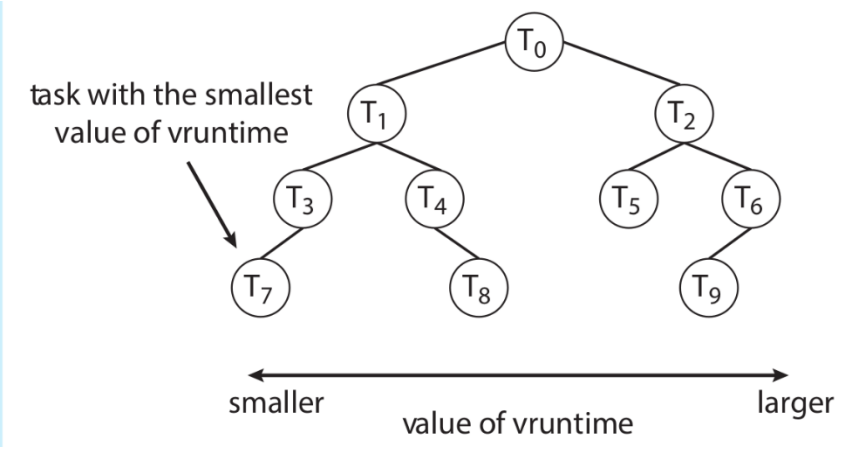
الگوریتم زمان‌بندی RR:

برنامه ای بنویسید که الگوریتم زمان‌بندی Round Robin را پیاده‌سازی کند. برای این کار مراحل زیر را انجام دهید:

۱. تعداد فرایندها را از کاربر دریافت کنید.
۲. زمان سرویس‌دهی هر فرایند را از کاربر دریافت کنید.
۳. کوانتوم زمانی را از کاربر دریافت کنید.
٤. ترتیب انجام فرایندها را نشان دهید.
٥. متوسط زمان انتظار هر فرایند را حساب کنید.

بخش دوم: الگوریتم زمان‌بندی CFS

زمان‌بند کاملا عادلانه (CFS) (Completely Fair Scheduler) در لینوکس الگوریتمی بهینه برای انتخاب کردن یک task (پردازه یا نخ) برای اجرا در مرحله بعدی است. برای پیاده‌سازی صف مورد استفاده در این الگوریتم از درخت قرمز_سیاه (Red Black Tree) که درختی دودویی است استفاده می کنیم. کلیدهای آن‌ها را مقادیر vruntime تشکیل می‌دهند. vruntime مخفف virtual runtime است و نشان دهنده ی میزان زمانی است که وظیفه از پردازنده استفاده کرده یا زمانبندی شده است.



زمانی که وظیفه آماده‌ی اجرا می‌شود به درخت اضافه می‌شود. اگر هم یک وظیفه دیگر قابل اجرا نبود از درخت حذف می‌شود. به صورت کلی وظیفه‌هایی که میزان کمتری به آن زمان برای اجرا اختصاص داده شده است در سمت چپ درخت قرار می‌گیرد و وظیفه‌هایی که زمان بیشتری به آنها اختصاص داده شده است به سمت راست هدایت می‌شوند. بر اساس مشخصات درخت‌های جست‌وجوی دو-دویی چپ‌ترین برگ کمترین مقدار کلید را دارد که در مثال زمان‌بند کاملا عادلانه بدین معناست که وظیفه بیشترین اولویت را دارد.

همین‌طور چون درخت-قرمز سیاه درختی متناسب است جست و جو در آن برای پیدا کردن چپ‌ترین برگ به پیچیدگی زمانی O(N log N) نیاز دارد {در اینجا N تعداد گره‌های درخت است) همچنین برای بهینه بودن linux مقدار متغیر rb-leftmost را cache کرده و برای پیدا کردن وظیفه ای که بعد اجرا می‌شود از مقدار ذخیره‌شده استفاده می‌کند، با این‌کار به پردازه بعدی ای که می‌خواهد اجرا کند از مرتبه O(1) دسترسی دارد. برای اطلاعات بیشتر می‌توانید به مستندات هسته لینوکس مراجعه نمایید.

پیاده‌سازی الگوریتم:

در این آزمایش می‌خواهیم یک نمونه ساده شده از الگوریتم زمان‌بندی CFS پیاده‌سازی کنیم.

- پروژه را از این لینک clone کنید.

این پروژه شامل فایل‌هایی برای استفاده از red-black tree است، تعریف functionها و structها در فایل‌های با پسوند h و پیاده‌سازی functionها در فایل‌های با پسوند c پیاده‌سازی شده‌اند.

فایل‌های داخل پروژه و توضیحات مربوط به آن‌ها:

فایل‌های مربوط به داده ساختار red black tree داخل پوشه red_black_tree:

- فایل rb.h
  - تعریف ساختارهای rbnode، rbtree و declarationهای functionهای مورد استفاده در red-black tree.
- فایل rb.c
  - پیاده‌سازی functionهای تعریف شده در rb.h
- فایل rb_data.h
  - تعریف ساختار mydata و توابع مورد نیاز برای استفاده از red-black tree و ذخیره داده در آن.
- فایل rb_data.c
  - پیاده‌سازی functionهای تعریف شده در rb_data.h
- فایل rb_example.c
  - یک نمونه استفاده از red-black tree.
- فایل minunit.h
  - فایل کمکی برای پیاده‌سازی unit test عملکرد red-black tree.
- فایل rb_test.c
  - آزمون unit test برای عملکرد red-black tree پیاده‌سازی شده.
- فایل rb_test.sh

- پیاده‌سازی دستور compile کردن و test کردن red-black tree پیاده‌سازی شده.

فایل‌های اصلی پروژه:

- فایل proc.h
  - تعریف ساختار پردازه (process) و functionهایی برای ساخت (create_process)، اجرا برای یک تیک (run_process_for_one_tick)، اتمام کار (terminate_process) و چک کردن به اتمام رسیدن پردازه (is_terminated).
- فایل proc.c
  - پیاده‌سازی functionهای تعریف شده در proc.h.
- فایل cfs_scheduler.c
  - فایل پیاده‌سازی الگوریتم زمان‌بندی CFS.
- فایل cfs.sh
  - پیاده‌سازی دستور compile کردن و اجرا کردن الگوریتم زمان‌بندی CFS.

همانطور که در فایل proc.h مشاهده می‌شود، یک پردازه به شکل زیر تعریف شده:

```
typedef struct{
  int id;
  int vruntime;
  int residual_duration;
} process;
```

که صرفا شامل id، vruntime, residual_duration است. residual_duration، زمان باقی‌مانده اجرای پردازه است و با هر tick ای که پردازه اجرا می‌شود کاهش پیدا می‌کند. مقدار آن برای این استفاده می‌شود که مشخص شود اجرای پردازه به اتمام رسیده یا خیر، اما این مقدار در تصمیم الگوریتم زمان‌بندی تاثیری ندارد.

حال به سراغ پیاده‌سازی در فایل cfs_scheduler.c می‌رویم.

۱. تعداد پردازه‌ها با دستور define در ابتدای فایل مشخص کنید (تعداد فعلی ۳ است).

۲. تابع fill_process_array را که یک آرایه خالی می‌گیرد و آن را با پردازه‌ها پر می‌کند را به شکل زیر پیاده‌سازی کنید:

```
void fill_process_array(process* process_array[PROCESS_COUNT]){
   for(int i = 0; i < PROCESS_COUNT; i++){
       process_array[i] = create_process(2000 + i, // process id
                                         100 + i,  // current vruntime
                                         3);       // residual execution
duration of the process
   }
}
```

۳. حال تابع insert_one_process_to_rbtree که یک process را به red-black tree اضافه می‌کند را به شکل زیر پیاده‌سازی کنید (آرگومان اولی که makedata_with_object می‌گیرد معیار قرار گرفتن پردازه در red-black tree است که اینجا همان vruntime پردازه است):

```
void insert_one_process_to_rbtree(rbtree *rbt, process* proc){
   mydata *data = makedata_with_object(proc->vruntime, proc);
   if(rb_insert(rbt, data) == NULL){
       fprintf(stderr, "insert %d: out of memory\n", proc->id);
   }
}
```

۴. سپس تابع insert_processes_to_rbtree که یک آرایه از processها را به red-black tree اضافه می‌کند را به شکل زیر پیاده‌سازی کنید:

```
void insert_processes_to_rbtree(rbtree *rbt, process
*process_array[PROCESS_COUNT]){
   process *proc;
   for(int i = 0; i < PROCESS_COUNT; i++){
       proc = process_array[i];
       insert_one_process_to_rbtree(rbt, proc);
   }
}
```

۵. سپس تابع process_of_node را که process ای که درون rbnode مورد نظر از red-black tree قرار دارد را برمی‌گرداند را به صورت زیر پیاده‌سازی کنید:

```
process* process_of_node(rbnode* node){
   process* proc = (process *)((mydata *)(node->data))->object;
   return proc;
}
```

۶. حال به سراغ پیاده‌سازی تابع main میرویم، ابتدا آرایه ای از processها ساخته و آن را به وسیله تابع fill_process_array پر می‌کنیم:

```
process *processes[PROCESS_COUNT];
fill_process_array(processes);
printf("process array filled\n");
```

۷. سپس یک اشاره‌گر red-black tree ساخته و به وسیله تابع rb_create آن را پر می‌کنیم:

```
rbtree *rbt;
if ((rbt = rb_create(compare_func, destroy_func)) == NULL)
{
   fprintf(stderr, "create red-black tree failed\n");
}
else
{
   printf("tree created\n");
}
```

۸. سپس پردازه‌های ساخته شده درون آرایه را به داخل red-black tree اضافه می‌کنیم:

```
insert_processes_to_rbtree(rbt, processes);
printf("inserted processes to rbtree\n");
```

۹. حال یک اشاره‌گر به گره به نام node، یک اشاره‌گر به نام current_process و یک عدد صحیح که tick فعلی را نشان می‌دهد به نام current_tick تعریف می‌کنیم:

```
rbnode *node;
process *current_proc;
int current_tick = 0;
```

۱۰. حال در یک حلقه، تا وقتی که red-black tree ساخته شده گرهی داشته باشد، سمت چپ‌ترین گره (که دارای پردازه با کمترین مقدار vruntime است) را به وسیله macro به نام RB_MINIMAL می‌گیریم و پردازه داخل آن را استخراج کرده و اجرا می‌کنیم، و سپس آن گره را از درخت حذف می‌کنیم (چرا که پردازه vruntime بعد از اجرا تغییر می‌کند). حال اگر همچنان پردازه نیاز به ادامه اجرا داشت (residual_duration آن بزرگتر از ۰ بود) آن را دوباره به red-black tree اضافه می‌کنیم. همینطور current_tick را نیز در هر بار اجرای حلقه افزایش می‌دهیم:

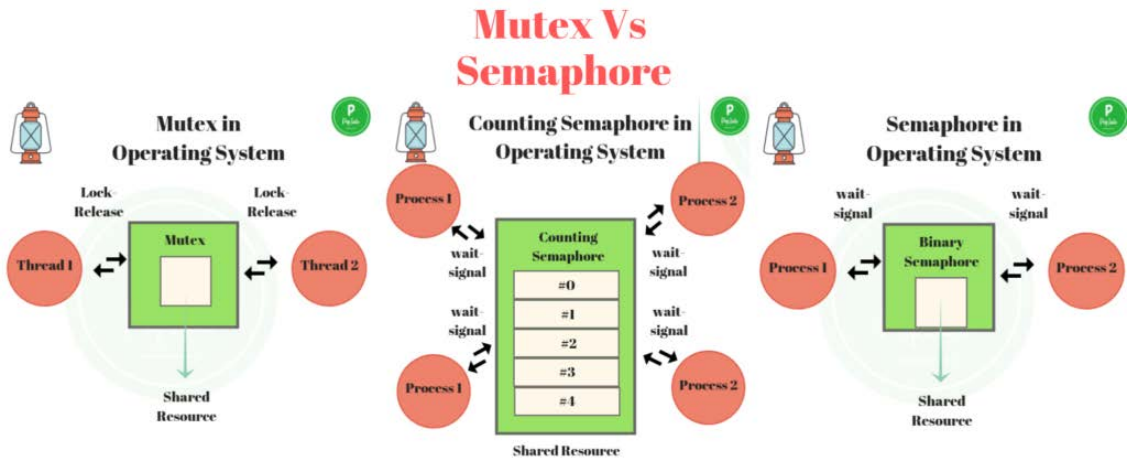
```
 while ((node = RB_MINIMAL(rbt)))
{
   printf("current_tick: %d\n", current_tick);
   current_tick++;
   current_proc = process_of_node(node);
   run_process_for_one_tick(current_proc);
   rb_delete(rbt, node, 0);
   if (!is_terminated(current_proc))
   {
      insert_one_process_to_rbtree(rbt, current_proc);
   }
}
```

در نهایت با اجرای فایل cfs.sh برنامه را اجرا کنید.

خروجی مورد انتظار آزمایش:

- انتظار می رود دانشجویان آزمایش را به طور کامل انجام دهند و برای توضیح کامل آن به مسئول آزمایشگاه آماده باشند.
- خروجی فایل خود را توضیح داده و علت ترتیب اجرای پردازه ها را توضیح دهید.

آزمایش نهم:
سمافور



هدف آزمایش

زمان‌بندی اجرای فرآیندها جهت استفاده
مناسب از منابع پردازشی سیستم عامل

آمادگی پیش از آزمایش:

مطالعه مفاهیم مرتبط با شرایط مسابقه

شرح آزمایش

مقدمه:

زمانی که فرآیندها به صورت همزمان اجرا می‌شوند و منابع بین آن‌ها مشترک است احتمال بروز شرایط مسابقه وجود دارد که در آن برنامه الزاماً در هر بار اجرا، پاسخ یکسانی تولید نخواهد کرد. برای جلوگیری از این مساله، نیاز به همگام‌سازی است. در این آزمایش هدف بررسی بیشتر این مساله است.

بخش اول:

سمافور یک متغیر عدد صحیح است که از طریق دو عملیات اتمی wait() و signal() دسترسی یا تغییر می‌یابد. در زبان C، عملیات‌های مربوطه به ترتیب با sem_wait() و sem_post() انجام می‌شوند. در ادامه برنامه‌ای برای همگام‌سازی فرایندها با استفاده از سمافورها آورده شده است تا پیاده‌سازی sem_wait() و sem_post() برای جلوگیری از شرایط مسابقه درک شود. برنامه ی زیر دو نخ ایجاد می‌کند یکی برای افزایش مقدار متغیر مشترک و دیگری برای کاهش مقدار آن. هر دو نخ از متغیر سمافور استفاده می‌کنند تا اطمینان حاصل شود که فقط یکی از نخ‌ها در بخش بحرانی خود در حال اجرا است:

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <unistd.h>
void *fun1();
 void *fun2();
 int shared=1; //shared variable
 sem_t s; //semaphore variable
 int main()
 {
 sem_init(&s,0,1);
 //initialize semaphore variable - 1st argument is address of variable, 2nd
is number of processes sharing semaphore, 3rd argument is the initial value
of semaphore variable
 pthread_t thread1, thread2;
```

```
 pthread_create(&thread1, NULL, fun1, NULL);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last updated
value of shared variable
 }
```

```
void *fun1()
{
    int x;
    sem_wait(&s); //executes wait operation on s
    x=shared;//thread1 reads value of shared variable
    printf("Thread1 reads the value as %d\n",x);
    x++;  //thread1 increments its value
    printf("Local updation by Thread1: %d\n",x);
    sleep(1); //thread1 is preempted by thread 2
    shared=x; //thread one updates the value of shared variable
    printf("Value of shared variable updated by Thread1 is: %d\n",shared);
    sem_post(&s);
}
```

```
void *fun2()
{
    int y;
    sem_wait(&s);
    y=shared;//thread2 reads value of shared variable
    printf("Thread2 reads the value as %d\n",y);
    y--;  //thread2 increments its value
    printf("Local updation by Thread2: %d\n",y);
    sleep(1); //thread2 is preempted by thread 1
    shared=y; //thread2 updates the value of shared variable
    printf("Value of shared variable updated by Thread2 is: %d\n",shared);
    sem_post(&s);
}
```

مقدار نهایی متغیر مشترک برابر با ۱ خواهد بود. وقتی که هر یک از نخ‌ها عملیات wait را اجرا می‌کند، مقدار متغیر سمافور S به صفر می‌رسد. بنابراین، نخ دیگر (حتی اگر نخ در حال اجرا را موقتی از اجرا خارج کند) قادر نخواهد بود تا عملیات wait را روی S به طور موفقیت‌آمیز اجرا کند. به این ترتیب، نمی‌تواند مقدار ناسازگار متغیر مشترک را بخواند. این امر اطمینان

می‌دهد که در هر لحظه فقط یکی از نخ‌ها در بخش بحرانی خود در حال اجرا است. خروجی برنامه به صورت زیر نشان داده شده است.

```
Thread1 reads the value as 1
Local updation by Thread1: 2
Value of shared variable updated by Thread1 is: 2
Thread2 reads the value as 2
Local updation by Thread2: 1
Value of shared variable updated by Thread2 is: 1
Final value of shared is 1
```

فرآیند، متغیر سمافور s را با استفاده از تابع sem_init() به ۱ مقداردهی اولیه می‌کند زیرا از سمافور دودویی استفاده شده است. اگر چندین نمونه از منبع در دسترس باشد، می‌توان از سمافور شمارشی استفاده کرد. سپس، فرآیند دو نخ ایجاد می‌کند. thread1 متغیر سمافور را با فراخوانی sem_wait() به دست می‌آورد. سپس، دستورات در بخش بحرانی خود را اجرا می‌کند. از تابع sleep(1) استفاده می‌کنیم تا نخ thread1 را موقتی از اجرا خارج کنیم و نخ thread2 را شروع کنیم. این سناریو شبیه‌سازی یک شرایط واقعی است. اکنون، هنگامی که thread2 تابع sem_wait() را اجرا می‌کند، قادر نخواهد بود این کار را انجام دهد زیرا thread1 قبلاً در بخش بحرانی خود قرار دارد. در نهایت، thread1 تابع sem_post() را فراخوانی می‌کند. حالا thread2 می‌تواند با استفاده از sem_wait() متغیر s را به دست آورد. این امر همزمانی بین نخ‌ها را تضمین می‌کند.

تمرین:

برنامه‌ای بنویسید که همگام‌سازی بین چندین نخ را برقرار کند. نخ‌ها سعی می‌کنند به منبعی به اندازه ۲ دسترسی پیدا کنند. به سوالات زیر پاسخ دهید.

- مقدار اولیه متغیر سمافور چیست؟
- چرا از تابع pthread_join() در برنامه استفاده می‌کنیم؟
- چرا پارامتر چهارم در pthread_create() برابر با NULL است؟
- اهمیت استفاده از sleep(1) در توابع fun1 و fun2 چیست؟
- چگونه از سمافورهای شمارشی استفاده کنیم؟

بخش دوم: مساله خوانندگان-نویسندگان

فرض کنید دو فرآیند reader و یک فرآیند writer وجود دارند که به ترتیب به خواندن مقدار بافر یا به روزرسانی آن می‌پردازند. بین این فرآیندها همانند روشی که در آزمایش قبل فراگرفتید یک حافظه مشترک در نظر بگیرید و در آن مقدار اولیه صفر را بنویسید. توجه داشته باشید که فرآیند writer دسترسی خواندن و نوشتن داشته باشد و فرآیند reader فقط دسترسی خواندن داشته باشد.

فرآیند writer با هر بار دسترسی به بافر مقدار موجود را یک واحد افزایش می‌دهد. writer بعد از دسترسی به بافر پیغامی چاپ می‌کند و در آن شماره فرآیند خودش (PID) و مقدار count را اعلام می‌کند. هر reader نیز به طور مداوم مقدار بافر را می‌خواند و در پیغامی شماره فرآیند خودش و مقدار count را اعلام می‌کند. توجه داشته باشید که هر دو reader می‌توانند با هم به بافر دسترسی داشته باشند. شرط پایان این است که مقدار count به یک مقدار بیشینه دلخواه برسد.

تمرین:

برنامه مربوطه را بصورت کامل نوشته و سپس اجرا کنید. به سوالات زیر پاسخ دهید:

- آیا مشکلی وجود دارد؟
- در صورت وجود ناهماهنگی چه راهکاری ارائه می‌کنید؟

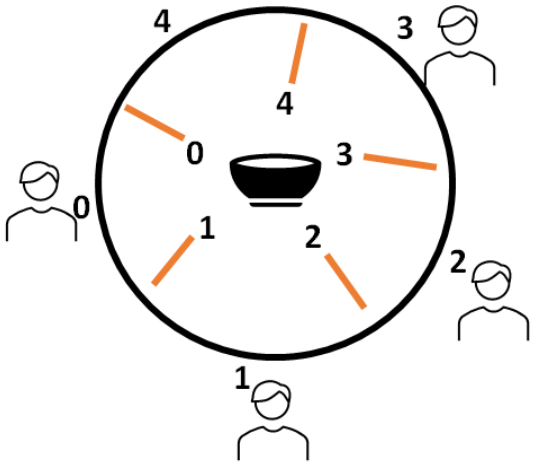
راهنمایی:

برای همگام سازی فرآیندهای reader و writer می‌توانید از روش‌های همگام‌سازی استفاده کنید. در این صورت وقتی اولین reader به بافر دسترسی می‌یابد باید آن را lock کند و وقتی آخرین reader کارش تمام شد lock را رها می‌کند. فرآیند writer زمانی می‌تواند مقداری بنویسد که فرآیند reader به بافر دسترسی نداشته باشد و تا اتمام عملیات نوشتن، فرآیند reader قادر به خواندن نیست.

بخش سوم: مساله فیلسوف‌های غذاخور

این یک مساله کلاسیک در مبحث همگام‌سازی فرآیندها است. این مسئله یک نمایش ساده از شرایطی است که تعدادی منبع در اختیار تعدادی فرآیند است و قرار است از پیش‌آمدن بن‌بست یا قحطی جلوگیری شود. میزی در نظر بگیرید که ۵ فیلسوف دور آن نشسته‌اند و ۵ چوب غذا (chopstick) برای غذا خوردن وجود دارد (بین هر دو صندلی یک چوب قرار دارد).

مسئله‌ی فیلسوفان غذاخور به این صورت است که پنج فیلسوف وجود دارند که دو کار انجام می‌دهند: فکر کردن و غذا خوردن. این فیلسوفان یک میز را به اشتراک می‌گذارند که برای هر کدام یک صندلی دارد. در مرکز میز، یک کاسه برنج قرار دارد و روی میز ۵ عدد چوب غذاخوری قرار داده شده است (به شکل زیر مراجعه کنید).



نکته) هدف این بخش، پیاده سازی این مسئله به زبان C است. بدین منظور هر چوب غذاخوری را با یک سمافور نمایش بدهید.

```
sem_t chopstick[5];
```

تمرین:

کد مسئله فیلسوفان غذاخور را پیاده‌سازی کنید و خروجی برنامه را به مدرس خود تحویل دهید. خروجی کد شما می‌تواند مانند زیر باشد:

```
Philosopher 0 wants to eat
Philosopher 0 tries to pick left chopstick
Philosopher 0 picks the left chopstick
Philosopher 0 tries to pick the right chopstick
Philosopher 0 picks the right chopstick
Philosopher 0 begins to eat
Philosopher 1 wants to eat
Philosopher 1 tries to pick left chopstick
Philosopher 3 wants to eat
Philosopher 3 tries to pick left chopstick
Philosopher 3 picks the left chopstick
Philosopher 3 tries to pick the right chopstick
Philosopher 3 picks the right chopstick
Philosopher 3 begins to eat
Philosopher 2 wants to eat
Philosopher 2 tries to pick left chopstick
Philosopher 2 picks the left chopstick
Philosopher 2 tries to pick the right chopstick
Philosopher 4 wants to eat
Philosopher 4 tries to pick left chopstick
Philosopher 0 has finished eating
Philosopher 0 leaves the right chopstick
Philosopher 0 leaves the left chopstick
Philosopher 1 picks the left chopstick
Philosopher 1 tries to pick the right chopstick
```

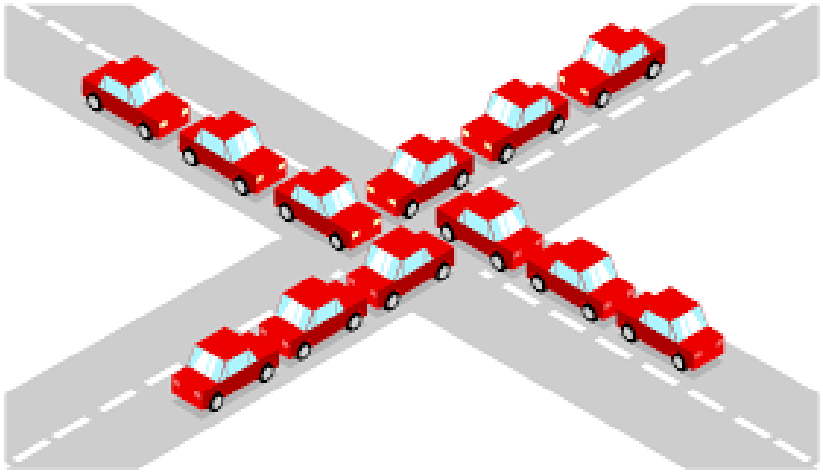
در اینجا فیلسوف (نخ) ۰ ابتدا سعی می‌کند غذا بخورد. بنابراین، او ابتدا سعی می‌کند چوب غذاخوری سمت چپ را بردارد که موفق می‌شود. سپس چوب غذاخوری سمت راست را برمی‌دارد. از آنجا که او هر دو چوب غذاخوری را برداشته است، فیلسوف ۰ شروع به غذا خوردن می‌کند. اکنون، به تصویر ابتدایی آزمایش مراجعه کنید. اگر فیلسوف ۰ شروع به غذا خوردن کند، این به این معنی است که چوب غذاخوری‌های ۰ و ۱ مشغول هستند، بنابراین فیلسوف‌های ۱ و ۴ نمی‌توانند غذا بخورند تا زمانی که فیلسوف ۰ چوب غذاخوری‌ها را پایین بگذارد. حالا خروجی را بخوانید، فیلسوف بعدی می‌خواهد غذا بخورد. او سعی می‌کند چوب غذاخوری سمت چپ (یعنی چوب غذاخوری ۱) را بردارد، اما موفق نمی‌شود زیرا چوب غذاخوری ۱ در حال حاضر در دست فیلسوف ۰ است. به همین ترتیب، می‌توانید بقیه خروجی را درک کنید.

سوال: آیا ممکن است بن بست رخ دهد؟ در صورت امکان چگونگی ایجاد آن را توضیح دهید.

خروجی مورد انتظار آزمایش:

- انتظار می رود دانشجویان آزمایش را به طور کامل انجام دهند و برای توضیح کامل آن به مسئول آزمایشگاه آماده باشند.
- هر سه تمرین را انجام داده و سوالات مرتبط با آن ها را پاسخ داده و نتایج را به مسئول آزمایشگاه نشان دهید.

آزمایش دهم:
شرایط مسابقه و بن‌بست



هدف آزمایش

آشنایی بیشتر با برنامه‌نویسی چند نخی ، ممانعت از شرایط مسابقه و اجتناب از بن بست

آمادگی پیش از آزمایش:

- مطالعه نحوه ایجاد یک نخ پوزیکس
- مطالعه نحوه استفاده از امکانات آماده نخ‌های پوزیکس برای جلوگیری از شرایط مسابقه

شرح آزمایش

توجه: آمادگی پیش از آزمایش برای این آزمایش بسیار ضروری است.

الگوریتم بانکداران:

در این آزمایش، یک برنامه چند نخی نوشته می‌شود که الگوریتم بانکداران را پیاده‌سازی کند. مشتری‌های متعددی منابع را از بانک درخواست می‌کنند و سپس پس می‌دهند بانکدار تنها در صورتی یک درخواست را اعطا خواهد کرد که سیستم در حالت امن باقی بماند. درخواستی که سیستم را در یک حالت ناامن باقی می‌گذارد رد می‌شود. در این جلسه ۳ موضوع چند نخی، ممانعت از شرایط مسابقه و اجتناب از بن بست را با هم ترکیب خواهید کرد. شبه کد بررسی امن بودن وضعیت به صورت زیر است:

```
1) Let Work and Finish be vectors of length 'm' and 'n' respectively
        Initialize: Work = Available
        Finish[i] = false; for i=1, 2, 3, 4...n

2) Find an i such that both
      a) Finish[i] = false
      b) Needi <= Work
if no such i exists goto step (4)

3) Work = Work + Allocation[i], Finish[i] = true
goto step (2)

4) if Finish[i] = true for all i, then the system is in a safe state
```

حال فرض کنید $Request_i$ آرایه درخواست‌های فرایند P_i است. $Request_i[j]$ به این معنی است که فرایند P_i تعداد k تا از منبع R_j درخواست دارد. کد درخواست منبع به صورت است:

1) If $Request_i$ <= $Need_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i$ <= Available
Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

Available = Available - $Request_i$
$Allocation_i$ = $Allocation_i$ + $Request_i$
$Need_i$ = $Need_i$ - $Request_i$

اگر وضعیت تخصیص منابع حاصل ایمن باشد، تراکنش تکمیل می‌شود و به فرآیند P_i منابع مورد نیاز اختصاص داده می‌شود. با این حال، اگر حالت ناامن باشد، P_i باید منتظر درخواست i باشد و وضعیت تخصیص منابع به حالت قبلی برمی‌گردد.

ALGORITHM:

1. Start the program.
2. Get the values of resources and processes.
3. Get the available value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. or not if we allow the request.
10. stop the program.
11.end

بانکدار:

بانکدار درخواست‌های n مشتری را برا m نوع منبع بررسی خواهد کرد. برای سادگی، فرض کنید ۵ نوع منبع داریم. بانکدار با استفاده از ساختمان داده‌های زیر، پیگیر منابع خواهد بود:

```
#define NUMBER_OF_RESOURCES 5
/* this maybe any values >= 0 */
#define NUMBER_OF_CUSTOMERS 5
/* the available amount of each resource */
int available[NUMBER_OF_RESOURCES];
/* the maximum demand of each customer*/
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
/* the amount currently allocated to each customer */
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
/* the remaining need of each customer */
int need [NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
```

مشتری:

تعداد n نخ مشتری ایجاد کنید که منابع بانک را درخواست و پس بدهند. مشتری‌ها به صورت مداوم، تعدادی تصادفی از منابع را درخواست و پس خواهند داد. درخواست‌های مشتری‌ها برای منابع به مقادیر متناظر آن‌ها در آرایه need محدود خواهند بود. بانکدار در صورتی با اعطای یک درخواست موافقت خواهد کرد که الگوریتم ایمنی مطرح شده در الگوریتم بانکداران حالت ایمن را نشان دهد. اگر درخواستی سیستم را در یک حالت امن باقی نگذارد، بانکدار آن را کنار خواهد زد. Prototype توابع درخواست و پس دادن منابع به صورت زیر هستند:

```
int request_resources(int customer_num, int request[]);
int release_resources(int customer_num, int request[]);
```

این دو تابع در صورت موفقیت، مقدار ۰ و در صورت عدم موفقیت مقدار ۱- را برمی‌گردانند. نخ‌های متعددی به صورت همروند، با استفاده از این دو تابع به داده‌های مشترکشان دسترسی خواهند داشت. بنابراین، دسترسی می‌بایست از طریق قفل‌های انحصار متقابل برای پیشگیری شرایط مسابقه کنترل شود. هر دوی APIهای نخ‌های پوزیکس و ویندوز، قفل‌های انحصار متقابل را فراهم می‌کنند .

پیاده‌سازی:

برنامه خود را با ارسال تعداد هر یک از انواع منابع بر روی خط فرمان احضار کنید. برای مثال اگر سه نوع منبع با ده نمونه از نوع اول، پنج نمونه از نوع دوم و هفت نمونه از نوع سوم وجود داشته باشد، برنامه خود را به صورت زیر احضار خواهید کرد:

```
./a.out 10 5 7
```

آرایه available با این مقادیر مقدار اولیه می‌گیرد. می‌توانید از کد زیر استفاده کنید. روش مناسب برای مقداردهی آرایه maximum را بیابید.

```
int main(int argc, char* * argv)
{
```

```
int available[6];
if (argc < 7)
{
printf("not enough arguments\n");
return EXIT_FAILURE;
}
for (int i = 0; i < 6; i++) {
available[i] = strtol(argv[i + 1], NULL, 10);
}
for (int i = 0; i < 6; i++) {
printf("av[%d]: %d\n", i, available[i]);
 }
return 0;
}
```

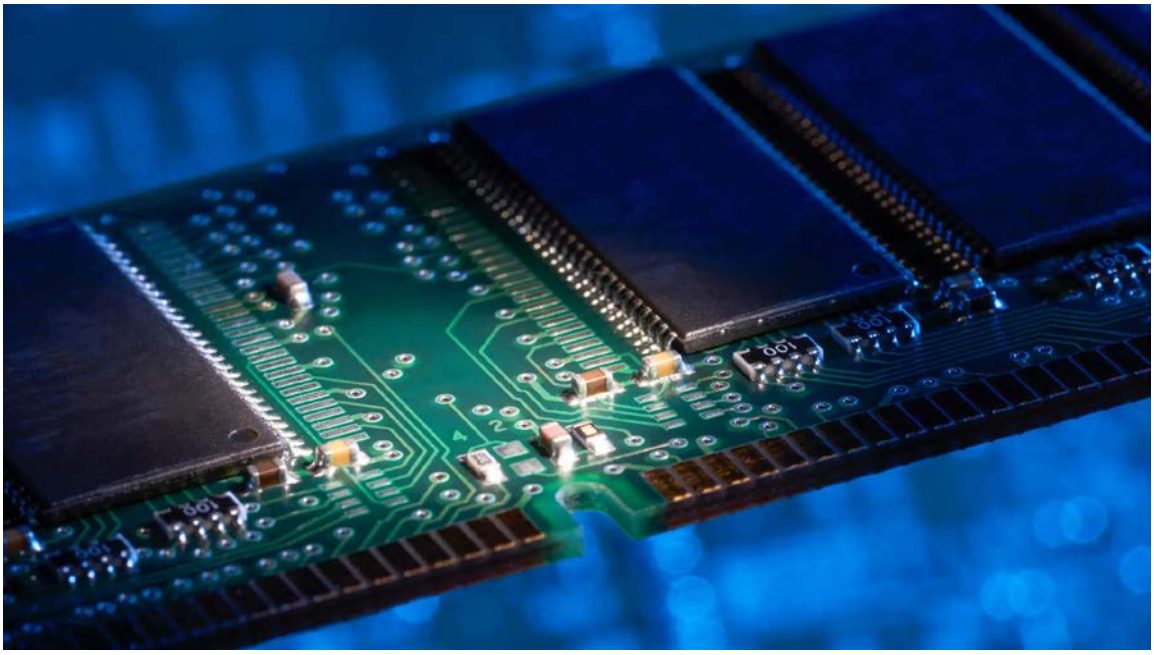
خروجی‌های مورد انتظار آزمایش:

- انتظار می‌رود بخش تمرین‌ها به صورت کامل توسط دانشجویان انجام شود و نتیجه به مدرس آزمایشگاه تحویل داده شود.
- تمامی مراحل انجام تمرین باید مرحله به مرحله توسط تمامی دانشجویان انجام شده و تمامی‌آن‌ها باید پس از انجام آزمایش قادر به توضیح مراحل مختلف باشند.

مراجع مطالعه/پیوست‌ها:

- https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
- https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
- https://stackoverflow.com/questions/9613934/pthread-race-condition-suspicious-behaviour

آزمایش یازدهم:
مدیریت حافظه



هدف آزمایش

آشنایی با نحوه مدیریت حافظه و نگاشت
صفحات فیزیکی به حافظه مجازی

آمادگی پیش از آزمایش:

شرح آزمایش

مقدمه:

مدیریت حافظه یکی از وظایف اصلی سیستم عامل است که در آن حافظه ی اصلی مدیریت می‌شود. فضای آدرس در واقع مجموعه ای از آدرس‌های منطقی می‌باشد که فرایندها در کدهای خود به آن ارجاع می‌دهند، به طور کلی قبل و بعد از تخصیص حافظه سه نوع آدرس داریم:

- آدرس‌های نمادین (Symbolic address): این نوع آدرس‌ها در سورس کدها استفاده می‌شوند شامل نام متغیر و ثابت‌ها از المان‌های اصلی فضای آدرس نمادین هستند.
- آدرس‌های نسبی (Relative address): در زمان کامپایل کد، کامپایلر آدرس‌های نمادین را به آدرس‌های نسبی تبدیل می‌کند.
- آدرس‌های فیزیکی (Physical address): در این بخش بازگذار یا همان loaderها این آدرس‌هارا زمانی که برنامه داخل حافظه ی اصلی بارگذاری شود، تولید می‌کند.

توجه کنید آدرس‌های فیزیکی و مجازی در زمان کامپایل و بارگذاری یکسان هستند اما در زمان اجرا تفاوت دارند. در این جا دو مفهوم مهم وجود دارد:

- به تمامی آدرس‌های منطقی که توسط برنامه به آن ارجاع داده شده است logical address space می‌گویند.
- به تمامی آدرس‌های فیزیکی مربوط به این آدرس‌های منطقی physical address space می‌گویند.

نگاشت آدرس مجازی به فیزیکی توسط واحد مدیریت حافظه یا Memory Management Unit که به اختصار به آن MMU می‌گویند و در واقع یک سخت افزار است، انجام می‌گیرد. در سیستم عامل لینوکس، امکان نگاشت یک فضای آدرس هسته به فضای آدرس کاربر وجود دارد. این کار، سربار ناشی از کپی کردن اطلاعات فضای کاربر به فضای هسته و بالعکس را از بین می‌برد.این ویژگی می‌تواند با استفاده از فراخوان سیستمی ()mmap در فضای کاربر مورد استفاده قرار گیرد. در ادامه به توضیح این تابع پرداخته می‌شود.

یکی از روش‌های مدیریت حافظه، صفحه بندی یا paging است که در آن آدرس به بلاک‌هایی با اندازه‌های یکسان شکسته می‌شود که به آن‌ها صفحه یا page گفته می‌شود. اندازه صفحه‌ها معمولاً ۴ کیلوبایت است، اما می‌تواند در برخی سکوها تا ۶۴ کیلوبایت نیز برسد.

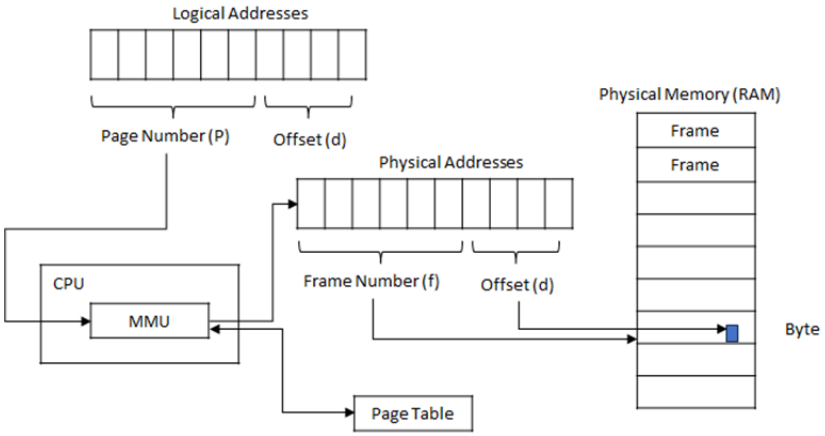
همانطور که در شکل ۱ مشاهده می‌نمایید می‌توان آدرس منطقی را به آدرس فیزیکی ترجمه نماییم. توجه کنید که آدرس منطقی با شماره ی صفحه (Page Number) و آفست (Offset) تعیین می‌شود.

Logical Address = Page number + Page offset

و آدرس فیزیکی با شماره ی قاب (Frame number) و شماره افست مشخص می‌شود.

Physical Address = Frame number + Page offset

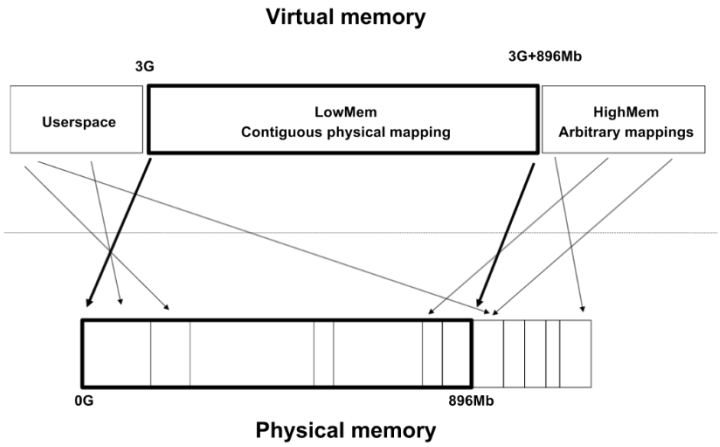
که frame number با استفاده از Page Map Table از یک نگاشت بک به یک از Page number بدست می‌آید. توجه کنید که می‌توانیم یک صفحه فیزیکی از حافظه یا Page Frame Number که به اختصار به آن PFN می‌گویند با استفاده از آدرس فیزیکی بدست بیاوریم(تقسیم آدرس فیزیکی بر اندازه صفحه). در ادامه در مورد این تابع در سیستم عامل لینوکس بحث خواهد شد.



شکل ۱

در سیستم عامل لینوکس به دلیل بهینه‌سازی عملکرد، فضای آدرس مجازی به دو بخش فضای کاربر و فضای هسته تقسیم می‌شود. به همین دلیل، فضای هسته شامل یک ناحیه حافظه نگاشت شده به نام lowmem است که به طور پیوسته در حافظه فیزیکی نگاشته (Map) شده و از پایین‌ترین آدرس فیزیکی ممکن (معمولاً صفر) شروع می‌شود. آدرس مجازی که lowmem در آن نگاشت شده توسط PAGE_OFFSET تعریف می‌شود. در یک سیستم ۳۲ بیتی، تمام حافظه موجود نمی‌تواند در lowmem نگاشت شود و به همین دلیل یک ناحیه جداگانه در فضای هسته به نام highmem وجود دارد که می‌تواند برای نگاشت حافظه فیزیکی استفاده شود.

حافظه‌ای که توسط تابع ()kmalloc تخصیص داده می‌شود در lowmem قرار دارد و از نظر فیزیکی پیوسته است. اما حافظه‌ای که توسط تابع ()vmalloc تخصیص داده می‌شود پیوسته نیست و در lowmem قرار ندارد. این حافظه یک ناحیه اختصاصی در highmem دارد.



سوال:

به موارد زیر پاسخ دهید:

- حافظه مجازی و حافظه فیزیکی چه تفاوتی با هم دارند؟
- چرا سیستم‌عامل‌ها از حافظه مجازی استفاده می‌کنند؟

فعالیت:

بررسی فایل **/proc/meminfo**:

- ابتدا با استفاده از دستور زیر، فایل /proc/meminfo را بررسی کنید:

```
cat /proc/meminfo
```

- این فایل شامل اطلاعاتی در مورد وضعیت حافظه سیستم است. مواردی مانند مجموع حافظه فیزیکی، حافظه استفاده‌شده و حافظه آزاد در این فایل نمایش داده می‌شود. اطلاعات موجود در این فایل را گزارش کنید و چند مورد از آن را به کمک این لینک بررسی و تحلیل کنید.

پیش از بحث درباره نحوه نگاشت حافظه، ابتدا به struct page که توسط Linux memory management subsystem استفاده می‌شود می‌پردازیم.

ساختار struct page:

ساختار struct page برای ذخیره اطلاعات مربوط به تمامی‌صفحات فیزیکی در سیستم استفاده می‌شود. تعدادی از توابعی که با این ساختار تعامل دارند عبارتند از:

- virt_to_page(): صفحه مرتبط با یک آدرس مجازی را برمی‌گرداند.
- pfn_to_page(): صفحه مرتبط با page frame number (PFN) را برمی‌گرداند.
- Page_to_pfn(): در این تابع page frame number مرتبط با یک struct page را برمی‌گرداند.
- page_address(): آدرس مجازی یک struct page را برمی‌گرداند؛ این تابع تنها برای صفحاتی که در lowmem هستند، قابل استفاده است.
- Kmap(): یک نگاشت در kernel برای یک صفحه فیزیکی دلخواه ایجاد می‌کند (می‌تواند از highmem باشد) و آدرس مجازی‌ای که برای reference مستقیم به صفحه استفاده می‌شود را برمی‌گرداند.

سوال:

- الگوریتم Paging چگونه کار می‌کند؟ بطور کلی توضیح دهید.
- الگوریتم Swapping چه نقشی در مدیریت حافظه ایفا می‌کند؟

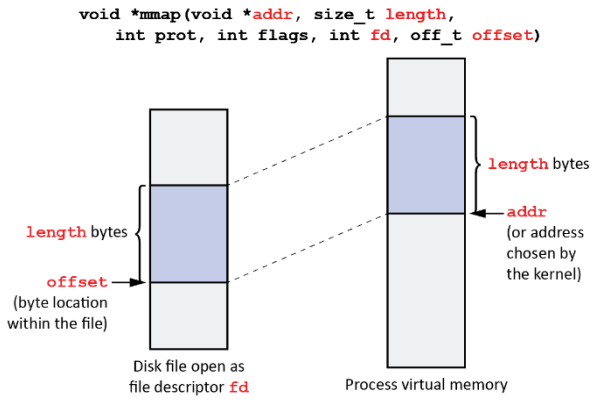
نگاشت حافظه (memory mapping)

نگاشت حافظه یکی از ویژگی‌های جالب در سیستم‌های یونیکس است. از دیدگاه یک درایور، این ویژگی امکان دسترسی مستقیم به حافظه فضای کاربر را برای دستگاه فراهم می‌کند. زمانی که یک برنامه اجرا می‌شود، محتوای برنامه در حال اجرا باید به فضای آدرس مجازی فرآیند منتقل شود. همین موضوع در مورد کتاب‌خانه‌های مشترکی که برنامه به آن‌ها لینک شده نیز صادق است. ولی این محتویات به طور واقعی به حافظه فیزیکی منتقل نمی‌شوند، بلکه تنها به فضای مجازی فرآیند لینک می‌شوند. سپس هنگامی که به قسمت‌های مختلف برنامه در حال اجرا ارجاعی داده می‌شوند، بخش مربوطه از فایل اجرایی به حافظه بارگذاری می‌شود. این فرآیند که در آن محتویات برنامه در حال اجرا به فضای آدرس مجازی فرآیند لینک می‌شود، نگاشت حافظه (Memory Mapping) نام دارد.

در سیستم‌عامل‌های لینوکس سیستم‌کال‌های mmap و munmap به برنامه‌ها اجازه می‌دهند تا کنترل دقیقی بر روی فضای آدرس خود داشته باشند. این سیستم‌کال‌ها می‌توانند برای اشتراک‌گذاری حافظه میان فرآیندها و نگاشت کردن فایل‌ها به فضای آدرس استفاده شوند. در این اینجا شما با این دو سیستم‌کال بیشتر آشنا می‌شوید و در نهایت با اجرای یک برنامه ساده به صورت عملی کارکرد آن‌ها را می‌بینید.

فراخوان سیستمی ()mmap

mmap() یک فراخوان سیستمی است که توسط یک پردازه ی سطح کاربر برای درخواست از هسته سیستم‌عامل به منظور نگاشت فایل‌ها یا دستگاه‌ها به حافظه (یعنی فضای آدرس) آن process استفاده می‌شود. نکته مهم این است که صفحات نگاشت‌شده در واقع تا زمانی که به آنها ارجاع داده نشود، وارد حافظه فیزیکی نمی‌شوند؛ بنابراین mmap() می‌تواند برای پیاده‌سازی بارگذاری تنبل (lazy loading) صفحات به حافظه (demand paging) مورد استفاده قرار گیرد. شکل زیر استفاده از سیستم‌کال mmap() را نشان می‌دهد:



فراخوان سیستمی mmap پارامترهای زیر را می‌پذیرد:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd,
 off_t offset);
```

با دستور man mmap هر یک از این شش آرگومان ورودی را بررسی کنید.

فعالیت: بررسی فایل **/proc/pid/maps**

فایل /proc/[pid]/maps اطلاعاتی درباره نگاشت حافظه یک process خاص را نشان می‌دهد. به جای [pid] باید شناسه فرآیند (PID) مورد نظر خود را قرار دهید. ابتدا با استفاده از دستور زیر لیست فرآیندهای در حال اجرا را مشاهده کنید:

```
ps aux
```

سپس شناسه فرآیند مورد نظر خود را انتخاب کرده و فایل مربوط به آن را با استفاده از دستور زیر مشاهده کنید:

```
sudo cat /proc/[pid]/maps
```

این فایل اطلاعاتی درباره بخش‌های مختلف حافظه فرآیند، شامل آدرس‌های مجازی و وضعیت آنها، ارائه می‌دهد. بخش‌های مختلف فایل را به کمک این لینک بررسی و توضیح دهید.

در صورت موفقیت، mmap() یک pointer به ناحیه نگاشت‌شده برمی‌گرداند. در صورت بروز خطا، مقدار MAP_FAILED برگردانده می‌شود (یعنی (void*)(-1)) و errno برای نشان دادن دلیل خطا تنظیم می‌شود.

فراخوان سیستمی munmap()

Munmap() یک فراخوان سیستمی است که برای "آزاد کردن" (unmap) حافظه‌ای که قبلاً با mmap() نگاشت شده است، استفاده می‌شود. فراخوان آن نگاشت حافظه را از فضای آدرس process فراخوان حذف می‌کند:

```
int munmap(void *addr, size_t length);
```

فراخوان سیستمی munmap() دو آرگومان دریافت می‌کند:

۱. addr: آدرس حافظه‌ای که باید از نگاشت مجازی process فراخوان آزاد شود. این آدرس باید مضربی از اندازه صفحه باشد.
۲. length: طول حافظه (تعداد بایت‌ها) که باید از نگاشت مجازی process فراخوان آزاد شود. این طول نیز باید مضربی از اندازه صفحه باشد.

در صورت موفقیت، munmap() مقدار ۰ را برمی‌گرداند. در صورت خطا، مقدار ۱- برگردانده می‌شود و errno برای نشان دادن دلیل خطا تنظیم می‌شود. اگر munmap() با موفقیت انجام شود، هرگونه دسترسی آینده به ناحیه حافظه‌ای که آزاد شده است منجر به خطای تفکیک حافظه (segmentation fault) یا SIGSEGV خواهد شد.

سوال: کابردهای استفاده از mmap() و munmap() چیست؟

فعالیت:

با استفاده از دستوراتی نظیر find، grep و locate آدرس و محل قرارگیری فایل‌هایی که مرتبط با mmap و munmap هستند را در سیستم‌عامل لینوکس پیدا کنید.

برای آشنا شدن با mmap و munmap و دیدن عملکرد آن‌ها بصورت عملی مراحل زیر را گام به گام انجام دهید.
یک فایل با نام mmap_example.c ایجاد کنید و کد صفحه بعد را در آن paste کنید.
برای کامپایل کردن کد Cدستور زیر را اجرا کنید:

```
gcc mmap_example.c -o mmap_example
```

حال برنامه کامپایل شده را با دستور زیر اجرا کنید:

```
./mmap_example
```

برنامه دارای سه وقفه است، یک وقفه قبل از اجرای ()mmap، یک وقفه بعد از اجرای آن، و در نهایت یک وقفه پس از اجرای ()munmap . شما باید پس از نمایش اولین وقفه یک صفحه پایانه(terminal) جدید باز کنید و با استفاده از دستور زیر PID فرآیند برنامه را پیدا کنید:

```
pgrep mmap_example
```

پس از پیدا کردن PID مربوط به فرآیند برنامه، دستور زیر را در صفحه پایانه ای که باز کرده‌اید اجرا کنید تا وضعیت حافظه نمایش داده شود:

```
cat /proc/[pid]/maps
```

پس از نمایش وضعیت، enter را در صفحه پایانه ی اول بزنید تا اجرای برنامه جلو برود و وقفه بعدی را چاپ کند. دوباره دستور مربوط به وضعیت حافظه را بزنید تا وضعیت حافظه تغییر یافته نمایش داده شود. دو مرحله قبلی را برای وقفه سوم و آخر نیز تکرار کنید، در نهایت با زدن enter اجرای برنامه خاتمه می‌یابد. خروجی cat /proc/[pid]/maps را قبل و بعد از اجرای ()mmap با هم مقایسه کنید.

سوال: چه تغییری رخ داده؟ بررسی کنید که بعد از اجرای ()munmap چه تغییری رخ می‌دهد.
با توجه به نتایج بدست آمده، توضیح دهید چه تفاوتی بین ()mmap و ()munmap وجود دارد؟

کد C مورد استفاده:

این برنامه فایلی را باز کرده و آن را به حافظه نگاشت می‌کند، سپس داده‌هایی در حافظه نگاشت شده می‌نویسد و در نهایت این نگاشت را پاک می‌کند.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <string.h>

int main()
{
   int fd;
   char *mapped_mem;
   size_t length = 4096; // Size of the mapping

   // Open file for reading and writing
   fd = open("testfile.txt", O_RDWR | O_CREAT, 0666);
   if (fd == -1)
```

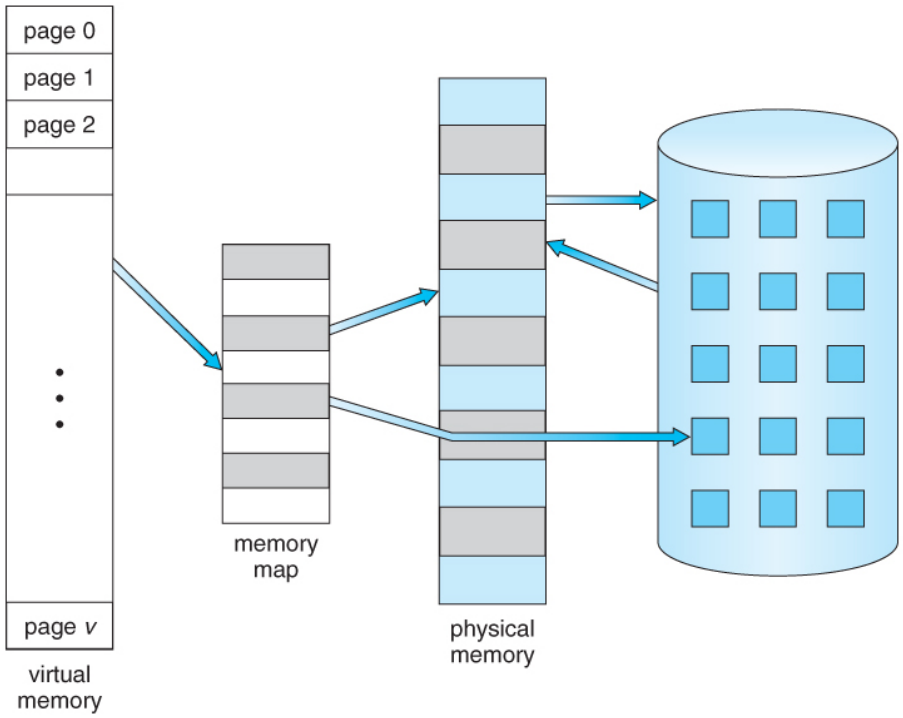
```
    {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    // Extend the file size to the size of the mapping
    if (ftruncate(fd, length) == -1)
    {
        perror("Error setting file size");
        close(fd);
        exit(EXIT_FAILURE);
    }

    // Pause to allow user to check memory map
    printf("Before executing mmap() - Press Enter to continue after
checking /proc/self/maps...\n");
    getchar();

    // Map file to memory
    mapped_mem = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
0);
    if (mapped_mem == MAP_FAILED)
    {
        perror("Error with mmap");
        close(fd);
        exit(EXIT_FAILURE);
    }

    // Write to the mapped memory
    strncpy(mapped_mem, "Hello, mmap!\n", length - 1); // Write a string to
the mapped memory
    mapped_mem[length - 1] = '\0';                    // Null-terminate
the string

    // Pause to allow user to check memory map
    printf("After executing mmap() - Press Enter to continue after checking
/proc/self/maps...\n");
    getchar();

    // Unmap the memory
    if (munmap(mapped_mem, length) == -1)
    {
        perror("Error with munmap");
        close(fd);
        exit(EXIT_FAILURE);
    }

    // Pause to allow user to check memory map
    printf("After executing munmapp() - Press Enter to continue after
checking /proc/self/maps...\n");
    getchar();

    close(fd);
    return 0;
```

```
}
```

خروجی‌های مورد انتظار آزمایش:

- انتظار می‌رود بخش سوال ها، فعالیت ها و آزمایش به صورت کامل توسط دانشجویان انجام شود و نتیجه به مدرس آزمایشگاه تحویل داده شود.

آزمایش دوازدهم:
جایگزینی صفحه در حافظه



هدف آزمایش

آمادگی پیش از آزمایش:

مرور آزمایش قبلی

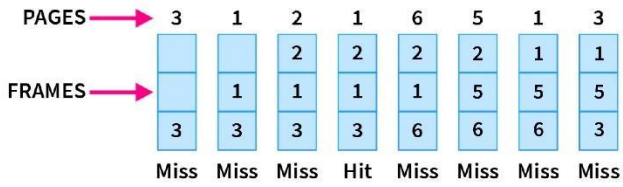
شرح آزمایش

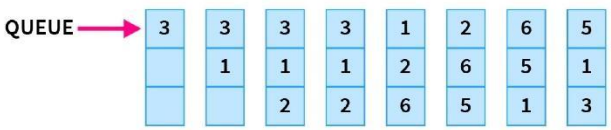
بخش اول:

الگوریتم‌های جایگزینی صفحه روش‌هایی هستند که در سیستم‌عامل‌ها برای مدیریت کارآمد حافظه زمانی که حافظه مجازی پر است استفاده می‌شوند. هنگامی که یک صفحه جدید باید در حافظه فیزیکی بارگذاری شود و فضای خالی وجود ندارد، این الگوریتم‌ها تعیین می‌کنند که کدام صفحه موجود را جایگزین کند. در این الگوریتم‌ها اگر صفحه مورد جستجو در میان frameها یافت شود، این فرایند به عنوان page hit شناخته می‌شود. اگر صفحه مورد جستجو در میان قاب‌ها یافت نشد، page fault رخ می‌دهد. اگر هیچ page frame ای آزاد نباشد، عملیات جایگزینی صفحه انجام می‌شود تا یکی از صفحات موجود در حافظه را با صفحه ای که reference آن باعث page fault شده است، جایگزین کند. در ادامه توضیح کوتاهی از چند الگوریتم داده می‌شود.

الگوریتم جایگزینی صفحه First in First out (FIFO)

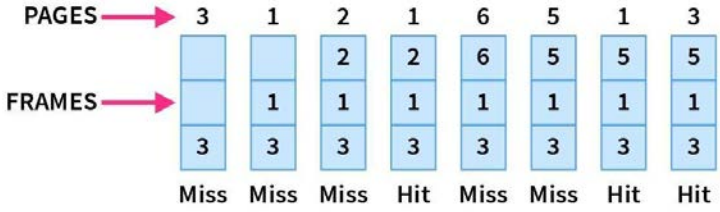
این ساده‌ترین الگوریتم جایگزینی صفحه است. در این الگوریتم، سیستم عامل تمام صفحات موجود در حافظه را در یک صف بررسی می‌کند که قدیمی‌ترین صفحه در جلوی صف قرار دارد. هنگامی که یک صفحه نیاز به جایگزینی دارد، صفحه‌ای که در جلوی صف قرار دارد برای حذف انتخاب می‌شود. مثال: رشته ارجاعات ۳,۱,۲,۱,۶,۵,۱,۳ را با ۳ page frame در نظر بگیرید. روند قرارگیری صفحات را در تصویر زیر مشاهده می‌کنید.





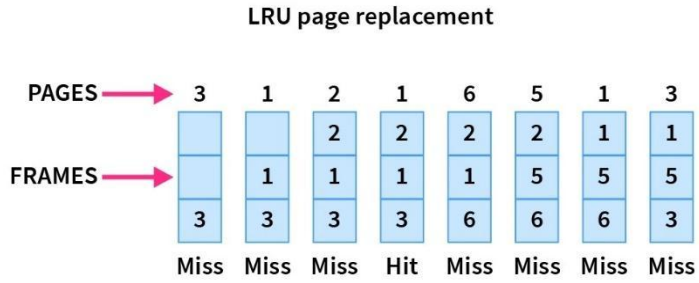
الگوریتم جایگزینی صفحه بهینه

جایگزینی صفحه بهینه بهترین الگوریتم جایگزینی صفحه است زیرا این الگوریتم منجر به کمترین تعداد خطاهای صفحه می‌شود. در این الگوریتم، صفحاتی که در آینده برای طولانی‌ترین مدت استفاده نخواهند شد جایگزین می‌شوند. مثال: رشته ارجاعات ۳,۱,۲,۱,۶,۵,۱,۳ را با ۳ page frame در نظر بگیرید. روند قرارگیری صفحات را در تصویر زیر مشاهده می‌کنید. وقتی صفحه ۶ می‌آید، صفحه ۲ حذف می‌شود چونکه در ادامه به ۱ و ۳ ارجاع وجود دارد اما به ۲ خیر.



الگوریتم جایگزینی صفحه Least Recently Used (LRU)

این الگوریتم ردیابی میزان استفاده از صفحات را در یک دوره زمانی مشخص نگه می‌دارد. این الگوریتم بر اساس اصل locality یک ارجاع کار می‌کند (این اصل بیان می‌کند یک برنامه تمایل دارد به مجموعه ای از مکان‌های حافظه به طور مکرر در مدت زمان کوتاهی دسترسی پیدا کند). بنابراین صفحاتی که در گذشته به‌شدت مورد استفاده قرار گرفته‌اند، احتمالاً در آینده نیز مورد استفاده قرار خواهند گرفت. مثال: رشته ارجاعات ۳,۱,۲,۱,۶,۵,۱,۳ را با ۳ page frame در نظر بگیرید. روند قرارگیری صفحات را در تصویر زیر مشاهده می‌کنید. وقتی صفحه ۶ می‌آید، صفحه ۳ حذف می‌شود چون به ۳ و ۲ یک ارجاع وجود دارد که چون ارجاع به ۳ قدیمی‌تر است حذف شده است.



فعالیت:

حال به انتخاب مسئول آزمایشگاه یکی از این الگوریتم‌ها را پیاده‌سازی کنید.

بخش دوم

برنامه‌های کاربردی چند نخی (multithreaded) روز به روز در حال پرکاربرد تر شدن هستند. با این حال، مدیریت حافظه آنها آسان نیست. در تخصیص‌دهنده‌های سنتی حافظه، نخ‌ها برای تخصیص یا آزاد کردن حافظه به طور همزمان با یکدیگر رقابت می‌کنند و این امر موجب گلوگاه‌ها (Bottleneck) می‌شود. این رقابت، عملکرد و مقیاس پذیری را کاهش می‌دهد.

تخصیص دهنده حافظه Hoard یک تخصیص دهنده حافظه سریع، مقیاس پذیر و کارآمد در حافظه است که بر روی طیف وسیعی از سکوها از جمله Linux، Mac OS و Windows ۱۰ و ۱۱ استفاده می‌شود. Hoard جایگزینی برای malloc است که می‌تواند عملکرد برنامه را به طور چشمگیری به ویژه برای برنامه‌های چند نخی بهبود بخشد.

معماری Hoard بر اساس مفهوم پشته یا همان heap است. هر پشته یک مخزن حافظه است که در آن تخصیص‌های حافظه انجام می‌شود. در ادامه به بررسی دقیق‌تر آن پرداخته می‌شود.

Superblockها:

سوپربلاک، بلوک بزرگی از حافظه پیوسته است که که به بلوک‌های کوچک‌تر با اندازه ثابت شکسته شده است. این بلوک‌ها همان چیزی هستند که برای جواب دادن به درخواست‌های حافظه اختصاص داده می‌شوند. سوپربلاک‌ها توسط هر پشته بر اساس میزان پر بودنشان گروه‌بندی می‌شوند. این امر باعث می‌شود که Hoard زمان زیادی را برای جستجوی تعداد زیادی از سوپربلاک‌ها هدر ندهد و بر اساس نیازش و با دانستن میزان پر بودن هر سوپربلاک انتخابش را انجام دهد.

انواع پشته در hoard:

۱. **پشته ی جهانی** (Global heap): Global heap باید توسط همه نخ‌ها به اشتراک گذاشته شود. این پشته سوپربلاک‌ها را ذخیره می‌کند و در صورت درخواست، آن‌ها را در اختیار هر پشته ی Per-Processor قرار می‌دهد.

۲. پشته per-processor: هدف از این نوع پشته این است که برای جلوگیری از مشاجره میان نخ‌ها، به هر نخ یک پشته محلی ارائه شده است. بنابراین، این پشته ها بخش کوچکی از حافظه را مدیریت می‌کنند که به نخ‌ها اجازه می‌دهد تا با کمترین تداخل سایر نخ‌ها، حافظه را تخصیص داده و آزاد کند. از آنجایی که هر نخ عمدتاً با پشته محلی خود تعامل دارد، نیاز به همگام سازی به حداقل می‌رسد و منجر به عملکرد بهتر در برنامه‌های چند نخی می‌شود.

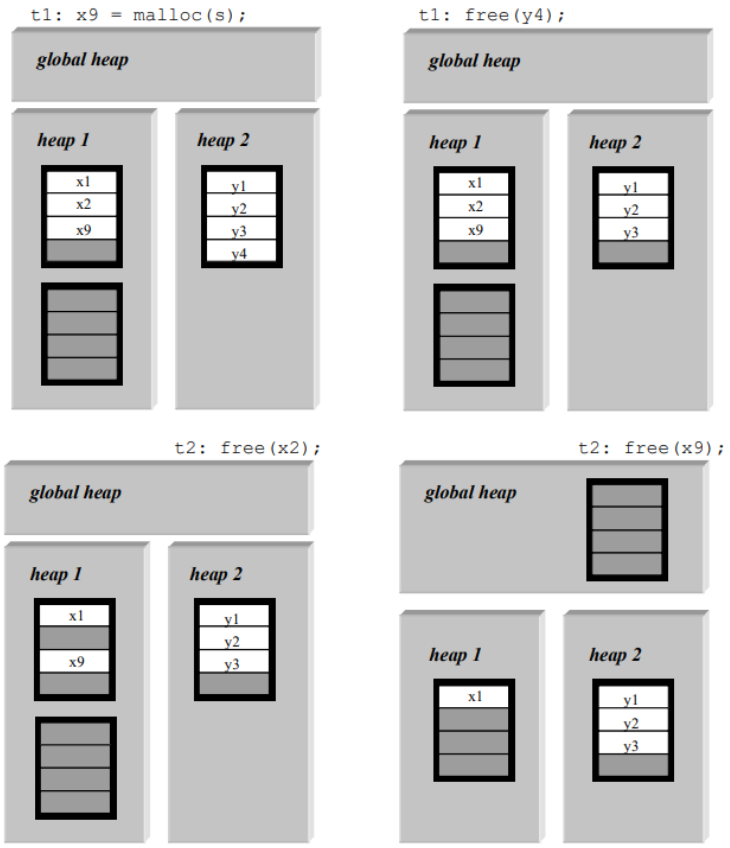
الگوریتم تخصیص حافظه:

هر زمان که یک نخ درخواست حافظه کند، بدون بررسی جزئیات مراحل زیر اجرا می‌شود:

نخ ابتدا سعی می‌کند حافظه را از Per-Processor Heap ش اختصاص دهد. سپس باید superblock مناسب را پیدا کند برای این امر سوپربلاک‌ها را بر اساس میزان پر بودنشان گروه‌بندی می‌کند و سعی می‌کند تا جایی که نیاز درخواست برطرف شود پر ترین سوپربلاک را انتخاب کند. اگر هیچ سوپربلاک مناسبی پیدا نشد نخ از global heap درخواست یک سوپربلاک جدید می‌دهد, آن را lock می‌کند و سوپربلاک را به پشته ی per-processor اضافه می‌کند.

یکی دیگر از اقدامات hoard این است که یک مقدار آستانه مشخص می‌کند و آن را empty fraction می‌نامد. زمانی که مقدار پر بودن superblock از empty fraction کمتر شود یک superblock از آن per-processor heap به global heap منتقل می‌کند. در ادامه یک مثال می‌بینیم تا با این مکانیزم‌ها بیشتر آشنا شویم.

مثال برای فهم بهتر جزئیات:



این شکل به نحو ساده شده، نحوه مدیریت سوپربلاک‌ها را نشان می‌دهد. برای سادگی، فرض می‌کنیم که دو نخ و پشته وجود دارد (نخ i به i heap نگاشت(map) می‌شود). در این مثال (که از سمت چپ بالا به سمت راست بالا، سپس از پایین چپ به پایین راست خوانده می‌شود)، empty fraction برابر ۴/۱ است. نخ ۱ کد نوشته شده با پیشوند ۱t و نخ ۲ کد نوشته شده با پیشوند ۲t را اجرا می‌کند.

در ابتدا، global heap خالی است، پشته ۱ دارای دو سوپربلاک (یکی تا حدی پر، یکی خالی) و پشته ۲ دارای یک سوپربلاک کامل است. تصویر اول پشته ها را پس از اینکه نخ ۱ ۹x را از پشته ۱ چون پرترین پشته است اختصاص داد نشان می‌دهد. در مرحله بعد، در نمودار بالا سمت راست، نخ ۱ ، ۴y را آزاد می‌کند، که در یک سوپربلاک در پشته ۲ است. از آنجایی که پشته ۲ هنوز ۴/۳اش پر است، Hoard سوپربلاک را از آن حذف نمی‌کند. در نمودار پایین سمت چپ، نخ ۲ ، ۲x را آزاد می‌کند، که در یک سوپربلاک متعلق به هیپ ۱ است. این آزاد شدن باعث نمی‌شود که پشته ۱ از آستانه خالی عبور کند، اما در تصویر آخر با آزاد شدن ۹x این اتفاق می‌افتد و می‌بینیم که Hoard ابر بلوک کاملا خالی موجود در پشته ۱ را به global heap منتقل می‌کند.

فعالیت:

۱- اکیدا توصیه می‌شود به مطالعه مقاله و کدهای این تخصیص‌گر حافظه در این بخش منابع مراجعه کنید و اطلاعات بیشتری کسب کنید.

۲- شبه‌کد تخصیص و آزادسازی حافظه در زیر آورده شده است. می‌توانید مفاهیم توضیح شده در بالا را در اینجا مشاهده کنید. به دنبال متغیرها و بخش‌هایی از شبه‌کد که برایتان آشنا نیست بگردید تا کاملا با مفهوم این دو الگوریتم آشنا شوید.

```
malloc (sz)
1.  If sz > S/2, allocate the superblock from the OS
      and return it.
2.  i ← hash(the current thread).
3.  Lock heap i.
4.  Scan heap i's list of superblocks from most full to least
      (for the size class corresponding to sz).
5.  If there is no superblock with free space,
6.    Check heap 0 (the global heap) for a superblock.
7.    If there is none,
8.      Allocate S bytes as superblock s
          and set the owner to heap i.
9.    Else,
10.     Transfer the superblock s to heap i.
11.     u_0 ← u_0 − s.u
12.     u_i ← u_i + s.u
13.     a_0 ← a_0 − S
14.     a_i ← a_i + S
15. u_i ← u_i + sz.
16. s.u ← s.u + sz.
17. Unlock heap i.
18. Return a block from the superblock.
```

```
free (ptr)
1.  If the block is "large",
2.    Free the superblock to the operating system and return.
3.  Find the superblock s this block comes from and lock it.
4.  Lock heap i, the superblock's owner.
5.  Deallocate the block from the superblock.
6.  u_i ← u_i − block size.
7.  s.u ← s.u − block size.
8.  If i = 0, unlock heap i and the superblock
      and return.
9.  If u_i < a_i − K * S and u_i < (1 − f) * a_i,
10.   Transfer a mostly-empty superblock s1
        to heap 0 (the global heap).
11.   u_0 ← u_0 + s1.u, u_i ← u_i − s1.u
12.   a_0 ← a_0 + S, a_i ← a_i − S
13. Unlock heap i and the superblock.
```

۳- همانطور که در توضیحات گفته شد Hoard یک سری از مشکلات رایج را با ارائه این الگوریتم حل کرده است. در ادامه چند مورد از آنها آورده شده است.

الف) تحقیق کنید که این مشکلات چه هستند و در چه شرایطی رخ می‌دهند.

ب) بررسی کنید که در hoard این موارد به چه شکلی مرتفع گردیده‌اند یا کاهش یافته‌اند.

- مشکل اشتراک نادرست (False Sharing)
- مشکل افزایش مصرف حافظه (Blowup)
- مشکل تکه تکه شدن حافظه (Fragmentation)

خروجی‌های مورد انتظار آزمایش:

- انتظار می‌رود بخش سوال ها، فعالیت ها و آزمایش به صورت کامل توسط دانشجویان انجام شود و نتیجه به مدرس آزمایشگاه تحویل داده شود.

مراجع مطالعه/پیوست‌ها:

مقاله:

https://people.cs.umass.edu/~emery/pubs/berger-asplos2000.pdf

کد:

https://github.com/emeryberger/Hoard

پایان