



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده مهندسی کامپیووتر

دستور کار آزمایشگاه سیستم‌های عامل

دکتر حمیدرضا زرندی
با همکاری مدرسین آزمایشگاه

نسخه ۲

۱۳۹۸ زمستان

بسم الله تعالى

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 طول ترم 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 طول ترم انجام می‌شود، در جلسه اول، توسط مدرس آزمایشگاه تعیین می‌شود، دانشجویان در هر جلسه به صورت تک نفره 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 صلاح‌الدید مدرس، ممکن است پروژه‌ای مدنظر قرار گیرد.

▪ قبل از شروع آزمایش هر گروه لازم است پیش گزارش تهیه کرده و قبل از شروع کلاس تحويل 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.

▪ از آنجایی که 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 تحويل داده شده	مجموع آزمایش‌ها حدود ۱۰ درصد
نمره پرسش‌های شفاهی/اکتبی 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 صلاح‌الدید مدرس آزمایشگاه)	به انتخاب مدرس آزمایشگاه

فهرست آزمایش‌ها

صفحه	عنوان آزمایش	موضوع	شماره آزمایش
۴	آشنایی با مقدمات لینوکس	Linux introduction	۱
۱۳	برنامه‌نویسی واحدهای هسته لینوکس	Kernel module programming	۲
۱۹	دستورنویسی در سیستم‌عامل	Bash scripting	۳
۲۸	فرآیندها و نخها	Thread and process	۴
۳۱	استفاده از مکانیزم‌های ارتباط بین فرآیندها	Inter-process communication	۵
۳۴	همگام‌سازی فرآیندها	Synchronization	۶
۳۶	بنیست و الگوریتم بانکداران	Deadlock	۷
۳۹	شبیه‌سازی الگوریتم‌های زمان‌بندی	Scheduling	۸
-	پروژه موضوعی از طرف مدرس	Final project	۹

Linux introduction

۱

آشنایی با مقدمات Linux

موضوع:	
شماره آزمایش:	
عنوان:	
هدف:	

آشنا شدن با محیط سیستم‌عامل لینوکس

آمادگی پیش از آزمایش:

شرح آزمایش:

بخش اول: تاریخچه

در سال ۱۹۷۱، سیستم‌عامل یونیکس (Unix) به دست تعدادی از مهندسان شرکت تلفن و تلگراف آمریکا (AT&T Corp.) توسعه پیدا کرد. این سیستم‌عامل که هرساله پیشرفت‌تر می‌شد، چندان ارزان نبود و همه نمی‌توانستند از آن استفاده کنند. در سال ۱۹۸۴ میلادی، ریچارد استالمان (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، هسته‌ای 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 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 زمانیه توسعه هسته سیستم‌عامل، لینوس تصمیم گرفت خودش دست به کار شود. در ۲۵ اوت 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. این گونه بود که او اولین نسخه از سیستم‌عامل لینوکس را سپتامبر همان سال منتشر کرد. دومین نسخه آن به فاصله کمی در 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 به تنهایی یک سیستم‌عامل مستقل محسوب می‌شود؟

لینوکس چیست؟

از دید فنی، لینوکس تنها نامی است برای هسته سیستم‌عامل و نه کل آن. دلیل این تعریف‌های گوناگون از لینوکس، به دلیل ماهیت انعطاف‌پذیر آن است. کمی بعد از عرضه این سیستم‌عامل، توروالدز تصمیم گرفت که به پروژه گنو بپیوندد. با این کار به سرعت توسعه لینوکس افزوده شد و توزیع‌های مختلفی ظاهر شدند. توزیع‌ها مجموعه‌ای از ابزارها هستند که برای رسیدن به

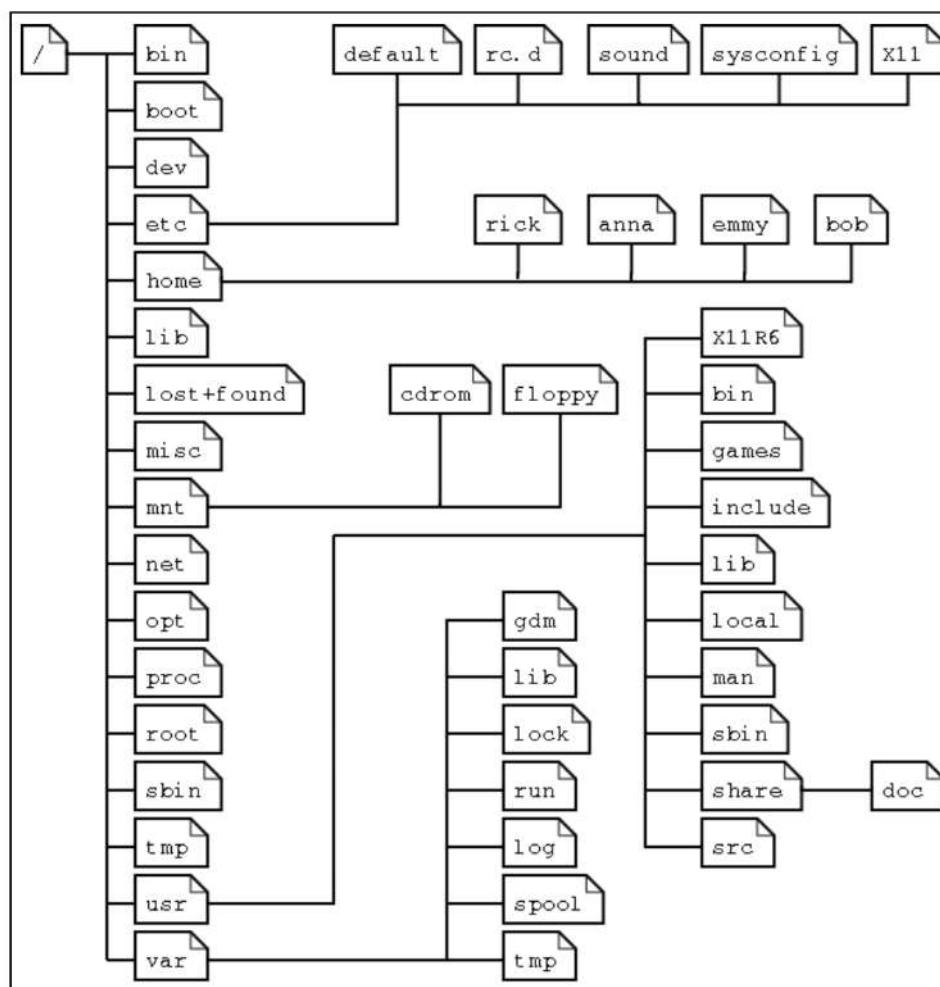
اهداف مختلف در کنار هم قرار می‌گیرند و از هسته لینوکس استفاده می‌کنند. به همین خاطر، لغت لینوکس را به سیستم‌عامل‌هایی اطلاق می‌کنند که از ترکیب‌بندی لینوکس (به عنوان هسته سیستم‌عامل) با نرم‌افزارهای آزاد و متن‌باز به دست می‌آیند. در صورتی که بنیاد نرم‌افزارهای آزاد تأکید دارد که از چنین سیستم‌عامل‌هایی، با عنوان گنو/لینوکس یاد شود. در این میان، سؤالی 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 به روز از سیستم‌عامل لینوکس را روی یک ماشین مجازی (یا حقيقی) را یاد گرفته و به صورت عملی انجام دهند.

بخش سوم: فایل سیستم لینوکس

ساختار فایل‌ها در سیستم یونیکس برای راحتی به صورت درختی در نظر گرفته می‌شود. در یک سیستم استاندارد شما زیر وجود دارد.



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

بخش چهارم : مدیریت فایل‌ها

برای شروع این بخش لازم است پایانه لینوکس را باز کنید. برای این کار می‌توانید از کلید 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 هر کدام کار خاص خود را انجام می‌دهند. در زیر لیست سوییچ‌ها قابل مشاهده است.

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

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 دستور cp و mv	
قبل از رونویسی ...	-f
قبل از رونویسی ...	-i
از فایل‌های رونویسی شده پشتیبان تهیه می‌کند	-b
صفات را حفظ می‌کند	-p

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	
برای پاک کردن دایرکتوری‌ها و محتوای داخل آن‌ها به صورت بازگشتی	-r,-R
حذف کردن به صورت اجباری	-f
قبل از هر حذف از کاربر سؤال می‌کند	-i

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

اگر دایرکتوری خالی نباشد برای پاک کردن آن از دستور rm -r dir_name استفاده می‌شود که در مورد ۴ گفته شد.

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

به معنی تمامی رشته‌ها	*
به معنی تمامی تک حرف‌ها است	?
تطابق می‌دهد با A, B و C	[ABC]
تطابق می‌دهد با حروف از a تا k	[a-k]
تطابق می‌دهد همه ارقام و همه حروف را	[0-9a-z]
به معنی هر حرف به جز x است	[!x]

می‌توان در دستوراتی که تا الان معرفی شده استفاده کرد. مثال:

```
rm *
cp * directory
cp * [a-f] directory
ls n*
ls t?
```

۸. دستور touch برای تغییر دادن تاریخ و زمان (Timestamp) به کار می‌رود و اگر فایل موجود نباشد آن را ایجاد می‌کند:
touch [options] file

لیست سوییچ‌های دستور touch

فقط زمان دستیابی (Access time) تغییر کند	-a
اگر فایل موجود نبود فایلی جدید تولید نکند	-c
رشته‌ای که پس از آن می‌آید را پارس کرده و به جای زمان فعلی استفاده می‌کند	-d
فقط زمان تغییر (Modification time) (Modification time) تغییر کند	-m
از زمان‌های فایل به جای زمان فعلی استفاده کند	-r
فایلی با زمان مشخص تولید کند	-t

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 محتوی فایل تغییر کرده است.

۳. Change time: آخرین زمانی که ابر داده (meta data) فایل (مانند permissions) تغییر کرده است.

برای مثال استفاده‌های مختلفی از این دستور در جدول زیر قابل مشاهده است:

```
touch filename
touch -d 10am filename
touch -d 13:50 filename
touch -d "yesterday 9pm" filename
touch -r referenceFile targetFile
```

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

۹. دستور find برای جستجو به صورت سلسله مراتبی استفاده می‌شود:

لیست سوییچ‌های دستور find

دنبال الگویی که پس از این سوییچ می‌آید، می‌گردد.	-name
فرقی با بخش بالایی ندارد به جز اینکه به کوچک یا بزرگ بودن حساس نیست	-iname
جستجوی دایرکتوری	-type d
جستجوی فایل	-type f
برای جستجو بر اساس حجم فایل استفاده می‌شود. + به معنی بزرگتر از N و - به معنی کوچک‌تر از N است. اگر عدد خالی باید به معنی بلوک است و با استفاده از c برای کاراکتر، G برای گیگابایت و ... می‌توان حجم را معلوم کرد.	-size +N/-N
برای جستجوی فایل یا دایرکتوری خالی استفاده می‌شود.	-empty

برای جستجوی فایل‌هایی که ۲۴*n ساعت قبل خوانده شده است.	-atime n
برای جستجوی فایل‌هایی که ۲۴*n ساعت قبل متأخرین تغییر کرده است.	-ctime n
برای جستجوی فایل‌هایی که ۲۴*n ساعت قبل محتوی آن تغییر کرده است.	-mtime n
برای جستجوی فایل‌هایی که n دقیقه قبل خوانده شده است.	-amin n
برای جستجوی فایل‌هایی که n دقیقه قبل متأخرین تغییر کرده است.	-cmin n
برای جستجوی فایل‌هایی که n دقیقه قبل محتوی آن تغییر کرده است.	-mmin n

مثال:

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

نکته: حال 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 به طوری 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 فشرده‌سازی، نسخه اصلی آن 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

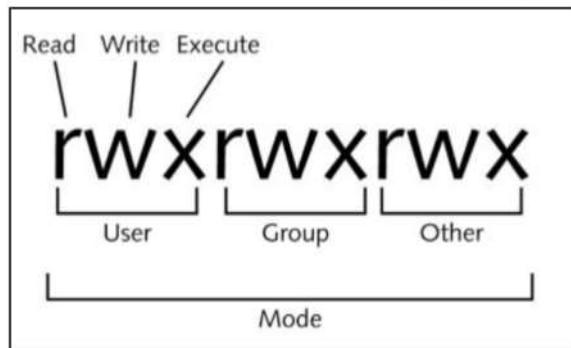
هر فایل شامل سه قسمت مجوز است:

الف) User permissions: مربوط به مالک فایل است.

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

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

هر دسته می‌توانند 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:

الف) u = user

ب) g = group

ج) o = others

د) a = all

عملیات:

الف) set(=)

ب) remove(-)

ج) give(+)

مجوزها:

الف) read(r)

b) write(w)

c) execute(x)

مثال:

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

همچنین مدل دیگری برای این کار وجود دارد که در شکل زیر قابل مشاهده است:

Octal mode - Permissions		
4 User 7	2 Group 7	1 Other 7
r	w	x
$4 + 2 + 1 = 7$		
Mode (one section only)		Corresponding Number
rwx	$4 + 2 + 1 = 7$	
rwx-	$4 + 2 = 6$	
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-
```

تمرین‌ها:

۱. دایرکتوری داخل میز کاری (Desktop) بسازید و تمامی مجوزهای آن را به گونه‌ای تغییر دهید که فقط شما و اعضای گروه بتوانند بنویسند، بخوانند و در آن جستجو 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 چه کاری 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).

خروجی‌های مورد انتظار آزمایش:

- انتظار می‌رود بخش تمرین‌ها به صورت کامل توسط دانشجویان انجام شود و نتیجه به مدرس آزمایشگاه تحويل داده شود.

مراجع مطالعه/پیوست‌ها:

1. R. Smith, LPIC-1, 3rd ed. Indianapolis, Indiana: John Wiley & Sons, 2013.

2. <https://jadi.gitbooks.io/lpic1/content>

Kernel module programming	موضوع:
۲	شماره آزمایش:
برنامه‌نویسی واحدها هسته لینوکس	عنوان:
	هدف:

آشنا شدن با نحوه نوشتن انواع واحدهای هسته و اجرای آن‌ها روی هسته

آمادگی پیش از آزمایش:

شرح آزمایش:

ماژول‌های هسته لینوکس:

در این جلسه چگونگی ایجاد یک ماژول هسته و بارگذاری آن روی هسته لینوکس آموزش داده می‌شود. می‌توان برای نوشتن این برنامه‌های C از یک ویرایشگر استفاده شود، لازم است که از یک برنامه ترمینال برای کامپایل برنامه‌ها استفاده کرده و فرمان‌ها را از طریق خط فرمان برای مدیریت ماژول‌های هسته وارد کنید. ماژول‌های هسته در آدرس /lib/modules قرار گرفته‌اند و با پسوند .ko. و در نسخه‌های قدیمی‌تر از ۲.۶ با پسوند .0. مشخص می‌شوند.

حسن نوشتن ماژول‌های هسته، این است که یک روش آسان برای تعامل با هسته ایجاد شود، لذا این امکان فراهم است که برنامه‌ای نوشته شود تا مستقیماً توابع هسته را فراخوانی کند. از آنجایی که این برنامه‌ها در هسته بارگزاری می‌شود، هر خطایی در کد برنامه می‌تواند باعث خرابی سیستم شود. با این وجود، بهتر است از ماشین مجازی استفاده شود تا هر خطأ در بدترین حالت مستلزم راهاندازی مجدد سیستم شود.

بخش ۱ - ایجاد ماژول‌های هسته:

اولین بخش این جلسه شامل یک سری مراحل برای ایجاد و درج یک ماژول در هسته لینوکس است. می‌توان تمامی ماژول‌های هسته را که در حال حاضر بارشده‌اند، با فرمان زیر فهرست کنید:

Lsmod

این فرمان، ماژول‌های فعلی هسته را در سه ستون فهرست می‌کند که عبارت‌اند از: نام، اندازه و جایی که ماژول استفاده می‌شود. برنامه زیر یک ماژول هسته بسیار ساده را به تصویر می‌کشد که در موقع بارگذاری و برداشتن ماژول هسته، پیغام‌های مناسبی چاپ می‌کند.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
/* this function is called when the module is loaded*/
int simple_init(void)
{
    printk(KERN_INFO "Loading Module\n");
    return 0;
}
/* this function is called when the module is removed*/
```

```

void simple_exit(void)
{
    printk(KERN_INFO "Removing Module\n");
}
/* Macros for registering module entry and exit points.
*/
module_init(simple_init);
module_exit(simple_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("simple module");
MODULE_AUTHOR("SGG");

```

تابع simple_init()، نقطه ورود ماژول است که معرف تابعی است که موقع بارگذاری ماژول در هسته احضار می‌شود. به طور مشابه تابع simple_exit() نقطه خروج ماژول است که موقع حذف ماژول از هسته فراخوانی می‌شود. تابع نقطه ورود ماژول باید یک مقدار صحیح برگرداند، صفر معرف موقتی‌آمیز بودن عملیات و مقادیر دیگر معرف خطا است. تابع نقطه خروج ماژول، void برمی‌گردد. به هیچ‌یک از نقاط ورود یا خروج ماژول پارامتری ارسال نمی‌شود. دو ماکروی زیر برای ثبت نقاط ورود و خروج ماژول در هسته استفاده می‌شوند:

```

module_init()
module_exit()

```

توجه گردد که چگونه هر دو تابع نقاط ورود و خروج، تابع printk() را فراخوانی می‌کنند. تابع printk() معادل هسته تابع printf() است. هرچند خروجی آن به یک بافر سابقه هسته فرستاده می‌شود که محتوی آن می‌تواند توسط فرمان dmesg خوانده شود. فرق میان printf() و printk() در این است که printk() امکان می‌دهد یک پرچم الیت مشخص گردد تا مقادیر آن در فایل سرآیند <linux/printk.h> تنظیم شده است که به عنوان یک پیغام اطلاعاتی تعریف می‌شود.

خطوط آخر MODULE_AUTHOR(), MODULE_DESCRIPTION() و MODULE_LICENSE() معرف جزئیات مربوط به مجوز نرمافزار، توصیف ماژول و نویسنده است. این کار تجربه استاندارد در نوشتن ماژول‌های هسته به حساب می‌آید. برای کامپایل ماژول‌های فرمان زیر را در خط فرمان وارد کنید:

```
make
```

کامپایل، فایل‌های متعددی تولید می‌کند. فایل simple.ko معرف ماژول هسته کامپایل شده است. مرحله بعدی، درج این ماژول را در هسته لینوکس روشن می‌سازد.

بارگذاری و حذف ماژول‌های هسته:

ماژول‌های هسته با استفاده از فرمان insmod بارگذاری می‌شوند که به صورت زیر اجرا می‌شود:

```
sudo insmod simple.ko
```

به منظور بررسی اینکه ماژول بارگذاری شده است یا خیر، فرمان lsmod را اجرا می‌گردد و ماژول simple را جستجو می‌شود. توجه شود نقطه ورود ماژول در موقع درج ماژول در هسته احضار می‌شود. برای بررسی محتوی این پیغام در بافر سابقه هسته، فرمان زیر استفاده می‌شود:

```
dmesg
```

بایستی پیغام "Loading Module" را مشاهده گردد. بهمنظور برداشتن ماژول هسته، فرمان rmmod استفاده می‌شود:

```
sudo rmmod simple
```

با بررسی فرمان dmesg اطمینان از برداشته شدن ماژول، حاصل می‌شود. چون بافر سابقه هسته می‌تواند به سرعت پر شود، بهتر است که به تناب بافر را خالی شود. این کار می‌تواند به صورت زیر انجام می‌شود:

```
sudo dmesg - c
```

تمرین ۱:

مراحل بالا را دنبال کنید تا یک ماژول هسته را ایجاد، بارگذاری و بردارید. ضمن بررسی محتوی بافر سابقه هسته مطمئن شوید مراحل کار را به درستی انجام داده‌اید.

بخش ۲ - ساختمان داده‌های هسته

بخش دوم این جلسه شامل اصلاح ماژول هسته است، طوری که از ساختمان داده لیست پیوندی هسته استفاده می‌کند. هسته لینوکس چند نمونه از ساختمان داده‌های مختلف را پوشش می‌دهد که در این جلسه استفاده از لیست پیوندی دوطرفه چرخشی را که برای توسعه دهنده‌گان هسته فراهم است، بررسی می‌شود. آنچه در اینجا بحث می‌شود در کد اصلی لینوکس که در فایل سرآیند <linux/list.h> موجود است و با گذر از گام‌های زیرین، بررسی می‌شود. در ابتدا، باید یک struct شامل عناصری که در لیست پیوندی درج می‌شوند، تعریف گردد:

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

به عضو struct list_head list در فایل سرآیند <linux/types.h> تعریف می‌شود. هدف آن، گذاشتن لیست پیوندی در میان گره‌های سازنده لیست است. رکورد list_head کاملاً ساده بوده و فقط دو عنصر دارد، next و prev، که به گره‌های قبلی و بعدی در لیست اشاره می‌کنند. با گذاشتن لیست پیوندی در میان رکوردها، لینوکس مدیریت ساختمان داده را به یک سری توابع ماکرو ممکن می‌سازد.

درج عناصر در لیست پیوندی:

ماکروی LIST_HEAD یک شی birthday_list اعلان می‌کند که به عنوان اشاره‌گری به ابتدای لیست استفاده می‌گردد:

```
static LIST_HEAD(birthday_list);
```

این ماکرو متغیر birthday_list را که از نوع struct list_head است، تعریف و مقداردهی می‌کند. نمونه‌های را به صورت زیر ایجاد کرده و مقداردهی می‌شود:

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

تابع `kmalloc()` معادل هسته‌ای تابع سطح کاربری `malloc()` برای تخصیص حافظه می‌باشد، جز اینکه در اینجا، حافظه هسته تخصیص داده می‌شود. (پرچم GFP_KERNEL تخصیص معمول حافظه هسته را نشان می‌دهد) دقت داشته باشید برای استفاده از `kmalloc()` می‌بایست از کتابخانه `<linux/slab.h>` استفاده کنید. ماکروی `INIT_LIST_HEAD`، عضو `list` در `struct birthday` را مقدار اولیه می‌دهد. در ادامه، می‌توان این نمونه را با استفاده از ماکروی `list_add_tail()` به انتهای لیست پیوندی اضافه کرد:

```
list_add_tail(&person->list, &birthday_list);
```

پیمایش لیست پیوندی

پیمایش لیست مشمول استفاده از ماکروی `list_for_each_entry()` است که سه پارامتر زیر را می‌پذیرد:

- اشاره گری به رکوردی که پیمایش روی آن صورت می‌گیرد.
- اشاره گری به سر لیستی که پیمایش روی آن صورت می‌گیرد.
- نام متغیر شامل رکورد `list_head`

کد زیر این ماکرو را به تصویر می‌کشد:

```
struct birthday *ptr;
list_for_each_entry(ptr, &birthday_list, list) {
    /*on each iteration ptr points to the next birthday struct*/
}
```

تمرین ۲:

در نقطه ورود مازول، یک لیست پیوندی شامل پنج عنصر `struct birthday` ایجاد کنید. لیست پیوندی را پیمایش کنید و محتوای آن را به بافر سابقه هسته انتقال دهید. فرمان `dmesg` را احضار کنید تا مطمئن شوید که به محض بار شدن مازول هسته، لیست به درستی ایجاد می‌شود.

در نقطه خروج مازول، عناصر لیست را از لیست پیوندی حذف کرده و دوباره حافظه آزاد شده را به هسته برگردانید. باز هم فرمان `dmesg` را احضار کنید تا بررسی کنید به محض برداشتن مازول هسته، لیست حذف می‌شود.

خروجی‌های مورد انتظار آزمایش:

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

مراجع مطالعه/پیوست‌ها:

هدف:	آشنایی با دستور نویسی در سیستم عامل و خودکارسازی کارهای لازم در خط دستور
عنوان:	دستور نویسی در سیستم عامل
شماره آزمایش:	۳

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

آمادگی پیش از آزمایش:

آشنایی با فایل سیستم‌های لینوکس و دستورات مقدماتی ترمینال لینوکس

شرح آزمایش:

مفسر زبان دستورات است. shell یک پردازشگر ماکرو است که دستورات را اجرا می‌کند. اصطلاح پردازشگر ماکرو به معنای عملکرد است که در آن متن و نمادها برای ایجاد عبارات بزرگ‌تر گسترش می‌یابد.

Unix shell مترجم دستورات و یک زبان برنامه‌نویسی است. به عنوان یک مفسر دستور، Shell رابط کاربری مشتمل بر مجموعه‌ای غنی از سرویسهای GNU ارائه می‌دهد. ویژگی‌های زبان برنامه‌نویسی اجازه می‌دهد که این سرویس‌ها ترکیب شوند. فایل‌هایی که حاوی دستورات هستند می‌توانند ایجاد شوند و خودشان تبدیل به دستور شوند. این دستورات جدید همانند دستورات سیستم در دایرکتوری‌هایی مانند bin/ هستند، این امکان را ایجاد می‌کنند تا کاربران یا گروه‌ها محیط‌هایی شخصی را برای بهینه‌سازی کارهای معمول خود ایجاد کنند.

با نوشتن مجموعه‌ای از دستورات در یک فایل متنی می‌توان به جای اجرای تک‌تک دستورات در ترمینال لینوکس با اجرای فایل به این هدف دست یافت. اگر قالب این فایل به صورت sh. باشد و اسم فایل را Sample.sh فرض شود، می‌توان با دستور ./sample.sh/. این فایل را اجرا کرد. حالت دیگر آن است که از دستور bash filename در ترمینال استفاده شود. برای ایجاد این فایل لازم است در اولین خط عبارت #!/bin/bash را قرار گیرد تا مشخص شود مفسر این دستورات bash است.

همانطور که میدانید، همه چیز در یونیکس یک فایل است (به جز آنچه که بین فایل‌ها قرار دارد). یونیکس بین فایل‌ها مکانیزمی به نام جریان^۱ را تعیین می‌کند که به داده‌ها اجازه می‌دهد بیت به بیت از یک فایل به یک فایل دیگر حرکت کنند. جریان دقیقاً چیزی است که به نظر می‌رسد: یک رودخانه کوچک از بیتها که از یک فایل به دیگری میریزد. اگر چه شاید پل نام بهتری باشد زیرا برخلاف جریان (که یک جریان دائمی از آب است) جریان بیتها بین فایل‌ها نباید ثابت باشد و یا حتی در همه موارد استفاده شود. سه جریان استاندارد پایه‌ای همه فایل‌ها به شرح زیر 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 ارورهای خروجی از فایل

process > data file	redirect the output of process to the data file; create the file if necessary, overwrite its existing contents otherwise.
process >> data file	redirect the output of process to the data file; create the file if necessary, append to its existing contents otherwise.
process < data file	read the contents of the data file and redirect that contents to process as input.

```
#!/bin/bash
ls > filename
```

مقداردهی به متغیرها:

مانند هر زبان برنامه‌نویسی دیگر در این زبان هم می‌توان متغیر تعریف کرد و به آن مقادیری نسبت داد. بدین منظور به مثال‌های زیر دقت کنید و برای هر قسمت فایلی ایجاد کرده و آن را اجرا کنید. برای چاپ مقادیر 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 $a
a=`ls -l` # Assigns result of 'ls -l' command to 'a'
echo "$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 شده‌اند و می‌توان در کابردهای خاص از آنها استفاده کرد. مانند:

\$0 – \$1 – \$9 – \$# – \$\$ – \$USER –

که در هنگام اجرای فایل می‌توان به فایل ورودی داد مثال: bash samplefile 1 3
که در آن مقادیر ۱ و ۳ که با فاصله آمده‌اند آرگومان هستند و برای استفاده از این آرگومان‌ها باید از متغیرهای \$1 و \$2 استفاده کرد. مقدار سایر متغیرهای خاص را بباید. اگر بیش از ۱۰ آرگومان ورودی باشد، چگونه باید به مقدار ۱۰-امین آرگومان دست یافت؟

برای دریافت مقادیر مورد نیاز از کاربر در حین اجرای برنامه از دستور read استفاده می‌شود. به مثال زیر توجه کنید –sp و –sp چه امکانی را فراهم می‌کنند؟ 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 شیوه‌های مختلفی وجود دارد. به مثال‌های 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
```

عبارات شرطی: برای نوشتن شرط از قالب زیر پیروی کنید:

```
if [ ] then
elif [ ] then
else
fi
```

- * اگر چند شرط مختلف داشته باشیم می‌توان اینگونه آنها را استفاده کرد: [] && [] [] || []
- * برای مقایسه اعداد می‌توان از -eq -lt -gt استفاده کرد که در مثال زیر به کار رفته است:

```
var1=10
var2=20
if [ $var1 -gt $var2 ] then
    echo "$var1 is greater than $var2"
fi
```

عبارات چند حالتی: قالب دستور case

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

```
*)  
commands  
;;  
esac
```

برخی از 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 length 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 its 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: قالب حلقه while و مثالی از آن در ادامه آمده است:

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

```
for OUTPUT in $(Linux-Or-Unix-Command-Here)
do
command1
command2
command3
done
```

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

توابع: تعریف تابع بصورت زیر تعریف می‌شوند:

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

دقت شود قبل از تعریف تابع نمی‌توان از آن استفاده کرد. اگر در تابع از دستور return استفاده گردد 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
}
print_hello john
echo $?
```

خروجی‌های مورد انتظار آزمایش:

انتظار می‌رود دانشجویان پس از یادگیری مطالب فوق بتوانند به سوالات زیر پاسخ دهند و نتیجه را به مدرس ارائه دهند.

۱. دستنوشتی (اسکریپتی) بنویسید که دو عددی که به صورت آرگومان به آن داده شده را

الف- با هم جمع کند و نتیجه را اعلام کند

- ب- عدد بزرگتر را نمایش دهد.
- ج- اگر کاربر در وارد کردن ورودی ها اشتباه کرده بود راهنمای مناسبی چاپ کند.
۲. ماشین حسابی با استفاده از `case` طراحی کنید.
۳. برنامه‌ای بنویسید که به طور متوالی از کاربر عدد دریافت کند و عددی چاپ کند که ترتیب ارقامش معکوس باشد. مثلا ۵۶۷ را به صورت ۷۶۵ چاپ کند. سپس جمع ارقام آن را چاپ کند.
۴. برنامه‌ای بنویسید که در هنگام اجرا دو عدد `x,y` و اسم یک فایل را دریافت کند و در خروجی خط `x` ام تا `y` ام فایل مذکور را نمایش دهد.
۵. برنامه‌ای بنویسید که از کاربر یک عدد بین ۱ و ۳ دریافت کند و شکل مربوط به آن عدد را رسم کند.

1 22 333 4444 55555	- - - - -
--	--	--

سوال امتیازی:

ماشین حسابی برای اعداد حقیقی بنویسید.

مراجع مطالعه/پیوست‌ها:

1. <http://www.gnu.org/software/bash/manual/bashref.html>
2. <http://www-h.eng.cam.ac.uk/help/tpl/unix/scripts/node18.html>
3. <http://www.gnu.org/software/bash/manual/bashref.html>

Inter-process communication

۴ موضع:

شماره آزمایش:

عنوان:

هدف:

استفاده از مکانیزم‌های ارتباط بین فرآیندها

ایجاد ارتباط بین فرآیندها در سیستم‌عامل لینوکس

آمادگی پیش از آزمایش:

فرآیندهایی که همزمان در سیستم‌عامل اجرا می‌شوند، می‌توانند مستقل یا همکار باشند. یک فرآیند مستقل تحت تاثیر فرآیندهای در حال اجرا در سیستم نیست و بر آنها تاثیری نمی‌گذارد. فرآیند مستقل با هیچ فرآیند دیگری داده به اشتراک نمی‌گذارد. در مقابل فرآیند همکار با فرآیندهای دیگر در حال اجرا در سیستم داده به اشتراک می‌گذارد و می‌تواند از آنها تاثیر بگیرد و یا تاثیر بگذارد.

فراهم کردن محیطی که فرآیندها بتوانند در آن با یکدیگر همکاری کنند دلایل متعددی دارد:

۱. به اشتراک گذاری اطلاعات: از آنجا که چندین کاربر ممکن است به یک اطلاعات مشترک علاقه‌مند باشند (مثلًا یک فایل اشتراکی)، باید بتوان محیطی برای دسترسی همزمان به این اطلاعات فراهم کرد.

۲. بالا بردن سرعت محاسبات: اگر قرار باشد کاری سریع انجام شود، یک روش، تقسیم آن به چند زیربخش و اجرای موازی آنهاست.

۳. پیمانه‌ای بودن: اگر قرار به ساخت سیستم بصورت پیمانه‌ای باشد، کارهای مختلف سیستم را به فرآیندها یا ریسمان‌های جداگانه تقسیم می‌شود.

با استفاده از مکانیزم‌های ارتباط بین فرآیندها، نیازمندی‌های بالا فراهم می‌شود. دو مدل اساسی در این بحث وجود دارد:

۱. حافظه مشترک (shared memory)

۲. تبادل پیام (message passing)

برای انجام روش حافظه مشترک، آشنایی با قابلیت‌های زیر در سیستم‌های POSIX نیاز است. مورد ۱ و ۲ توضیح داده شده است، توضیحات مربوط به بخش ۳ تا ۵ را از کتاب درس مطالعه کنید.

۱. هر فرآیندی برای ایجاد یک حافظه مشترک، از فراخوانی سیستمی `shmget()` استفاده می‌کند. به مثال زیر توجه کنید:

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

در این مثال، اولین پارامتر مشخص کننده کلید (identifier) برای قطعه حافظه مشترک است، اگر این مقدار برابر با `IPC_PRIVATE` باشد، یک حافظه جدید ساخته می‌شود. دومین پارامتر اندازه این قطعه را مشخص می‌کند. سومین پارامتر نوع این حافظه اشتراکی را مشخص می‌کند که تعیین کننده نحوه استفاده از آن است و می‌تواند نوشت، خواندن یا هر دو باشد. در این مثال، فرآیندی که این حافظه اشتراکی را ایجاد می‌کند، اجازه خواندن یا نوشت دارد. 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.

۲. فرآیند هایی که قصد استفاده از حافظه اشتراکی را دارند باید آن را به فضای آدرس خود اضافه کنند. این کار با فراخوانی سیستمی shmat() انجام می‌شود. به مثال زیر دقت کنید:

```
shared_memory =(char*) shmat(id, NULL, 0);
sprintf(shared_memory, "Writing to shared memory");
shmdt(shared_memory);
shmctl()
```

ارتباطات در یک سیستم خادم-خدمتگذار (Client-Server)

علاوه بر مکانیزم‌های قبلی، در این بحث سه مکانیزم دیگر نیز مورد استفاده است:

۱. socket
۲. pipe
۳. remote procedure calls (RPCs)

در این آزمایش هدف آن است به طور خاص روی برنامه‌نویسی سوکت تمرکز گردد. تابع لازم برای برنامه‌نویسی سوکت در هر زبان برنامه‌نویسی با دیگری تفاوت دارد. برای برنامه‌نویسی در این آزمایش فقط می‌توانید از زبان C استفاده کنید.

شرح آزمایش:

بخش اول:

محیطی آماده کنید که دو فرآیند در آن وجود داشته باشند و از روش حافظه مشترک برای ارتباط استفاده کنند. (برای مثال می‌توانید دو فرآیند در نظر بگیرید که یکی مقداری بنویسد و دیگری آن را بخواند)

بخش دوم:

در این آزمایش باید یک برنامه کاربردی گفتگو (Chat application) را به وسیله زبان C پیاده‌سازی کنید. این برنامه دو بخش دارد: سرور و کاربر. سرور اطلاعات کاربران را نگهداری می‌کند و وقتی پیامی توسط یک کاربر فرستاده می‌شود با توجه به گروه مورد نظر این پیام را بین کاربران پخش می‌کند. راه ارتباط کاربران با یکدیگر، وارد شدن به یک گروه چت است.

راه اندازی 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:

join [groupId]: این کاربر را به گروه مشخص شده اضافه می‌کند

send [groupId] [message]: پیام کاربر را به گروه مشخص شده می‌فرستد

leave [groupId]: کاربر را از گروه مشخص شده حذف می‌کند

quit: کاربر کلا از برنامه خارج می‌شود

وقتی یک کاربر درخواست join می‌دهد، اگر عضو آن گروه باشد، درخواست او بی‌اثر می‌شود و اگر عضو نیست به آن گروه اضافه شود. یک کاربر می‌تواند عضو گروه‌های متعددی شود. وقتی یک کاربر پیامی به یک گروه می‌فرستد این پیام باید به همه اعضای آن گروه فرستاده شود. برای مثال message: clientName: تواند قالب مناسبی برای این کار باشد.

برای مثالی از ارتباط کلاینت و سرور به نمونه‌ی زیر توجه کنید.
کد مربوط به کلاینت:

```
#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 سرور:

```
#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 -- converts 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;
}

```

بخش سوم:

محیطی فراهم کنید که در آن دو فرآیند با استفاده از خط لوله به تبادل یک پیام متنی بپردازنند. 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.

خروجی‌های مورد انتظار آزمایش:

انتظار می‌رود دانشجویان پس از یادگیری مطالب فوق بتوانند موارد خواسته شده را انجام دهند و نتیجه را به مدرس ارائه دهند.

مراجع مطالعه/پیوستها:

Threads and processes

موضوع:

۵

شماره آزمایش:

فرآیندها و نخها

عنوان:

هدف:

برنامه‌نویسی چند فرآیندی و کشیدن نمودار توزیع نرمال

آمادگی پیش از آزمایش:

شرح آزمایش:

مقدمه:

در روش‌های تحقیقات علمی، بررسی نمونه‌ای و تحقیقات آماری (Sampling)، به فرآیندی گفته می‌شود که براساس آن انتخاب اعضايی از جامعه آماری صورت می‌پذیرد. این کار با هدف برآورد پارامتر جامعه و یا شناخت بیشتر از آن انجام می‌شود. اهمیت نمونه‌گیری را می‌توان صرفه‌جویی در زمان برای تهیه مشاهدات از جامعه آماری به منظور انجام تحقیق علمی دانست. معمولاً 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 فرآیند با استفاده از تولید زوج عدد های 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.

تعريف مسائله:

در این آزمایش هدف آن است که با استفاده از نمونه برداری، نمودار توزیع نرمال را ترسیم شود. در ابتدا یک ارایه با نام hist که ۲۵ خانه دارد بسازید. از این آرایه برای نگهداری نتایج آزمایش استفاده می‌شود. این ۲۵ خانه نمایندگان اعداد ۱۲ - تا $12 +$ هستند. فرآیند نمونه برداری به این صورت است که مقدار ابتدایی متغیر counter شما با مقدار صفر شروع می‌شود و شما باایستی 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 ارایه hist را افزایش دهید.

شرح کلی:

گام‌های زیر را انجام دهید:

۱. ابتدا کد برنامه‌ای 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	زمان اجرا
۵۰۰۰۰	۵۰۰۰۰

۲. حال برنامه‌ای بنویسید که با استفاده از `fork()` و یا `exec()` تعدادی فرآیند فرزند ایجاد شود و کارها را پخت 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) آموخته‌اید برای ارتباط بین فرآیند های فرزند و پدر استفاده کنید.
زمان اجرا برنامه خود را در جدول زیر گزارش دهید.

تعداد نمونه	زمان اجرا
۵۰۰۰۰	

۳. ایا این برنامه در گیر 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 زیر گزارش دهید.

تعداد نمونه	افزایش سرعت
۵۰۰۰۰	

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

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

```
    printf("\n");
}
}
```

خروجی‌های مورد انتظار آزمایش:

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

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.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html>
4. <https://www.youtube.com/watch?v=VJTFfIqO4TU>

موضوع:	Synchronization
شماره آزمایش:	۶
عنوان:	همگام‌سازی فرآیندها
هدف:	ایجاد هماهنگی بین چند فرآیند در حال اجرا

آمادگی پیش از آزمایش:

مطلوب مربوط به این بحث در کلاس درس گفته شده است. قبل از شروع آزمایش آن‌ها را مرور کنید و در صورت ابهام، قبل از شروع آزمایش، سوالات خود را از مدرس آزمایشگاه بپرسید.

شرح آزمایش:

زمانی که فرآیندها به صورت همزمان اجرا می‌شوند و منابع بین آن‌ها مشترک است احتمال بروز شراب 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: مساله خواندن‌گان-نویسندها را پیاده‌سازی کنید.

بدین منظور فرض کنید دو فرآیند reader و یک فرآیند writer وجود دارند که به ترتیب به خواندن مقدار بافر یا به روزرسانی آن می‌پردازنند. بین این فرآیندها همانند روشی که در آزمایش قبلاً فراگرفتیم یک حافظه مشترک در نظر بگیرید و در آن مقدار اولیه صفر را بنویسید. توجه داشته باشید که فرآیند writer دسترسی خواندن و نوشتمن 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.

برنامه مربوطه را بصورت کامل نوشته و سپس اجرا کنید. آیا مشکلی وجود دارد؟ در صورت وجود ناهمانگی چه راهکاری ارائه می‌کنید؟

راهنمایی: برای همگام‌سازی فرآیندهای reader و writer می‌توانید از روش‌های همگام‌سازی استفاده کنید. در این صورت وقتی اولین reader به بافر دسترسی می‌باشد 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 نوشتمن، فرآیند reader قادر به خواندن نیست.

بخش دوم: 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 پیش آمدن بن بست یا قحطی جلوگیری شود. میزی در نظر بگیرید که ۵ فیلسوف دور آن نشسته‌اند و ۵ چوب غذا برای غذا خوردن وجود دارد (بین هر دو صندلی یک چوب قرار دارد). هر فیلسوف مدتی تفکر می‌کند و وقتی گرسنه شد دو چوب غذا خوری لازم دارد تا از غذای وسط میز بخورد. فیلسوف در یک زمان مشخص می‌تواند فقط یک چوب بردارد و اگر چوب در اختیار فیلسوف کناری باشد، قاعده‌تاً نمی‌تواند آن را بردارد. پس از خوردن غذا چوب‌ها را روی میز می‌گذارد تا بقیه در صورت نیاز از آن‌ها استفاده کنند.

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

هدف این بخش، پیاده‌سازی این مسئله به زبان C است. بدین منظور می‌توانید از موارد زیر استفاده کنید.

```
pthread_t philosopher[5];
pthread_mutex_t chopstick[5];
```

برای اجرا کردن کدی که نوشته‌اید در ترمینال لازم است از دستوراتی مشابه دستورات زیر استفاده کنید.

```
gcc -pthread -o test1 test1.c
./test1
```

خروجی کد شما می‌تواند مانند زیر باشد.

```
philosopher 1 is thinking !!
philosopher 1 is eating using chopstick[0] and chopstick[1]!!
philosopher 5 is thinking !!
philosopher 3 is thinking !!
philosopher 3 is eating using chopstick[2] and chopstick[3]!!
philosopher 2 is thinking !!
philosopher 4 is thinking !!
philosopher 1 finished eating !!
philosopher 5 is eating using chopstick[4] and chopstick[0]!!
philosopher 2 is eating using chopstick[1] and chopstick[2]!!
philosopher 3 finished eating !!
philosopher 5 finished eating !!
philosopher 2 finished eating !!
philosopher 4 is eating using chopstick[3] and chopstick[4]!!
```

خروجی‌های مورد انتظار آزمایش:

انتظار می‌رود دانشجویان پس از انجام بخش‌های فوق نتیجه را به مدرس ارائه دهند.

مراجع مطالعه/پیوست‌ها:

Deadlock	موضوع:
۷	شماره آزمایش:
بن بست و الگوریتم بانکداران	عنوان:
	هدف:

پیاده‌سازی الگوریتم بانک دار ها

آمادگی پیش از آزمایش:

- مطالعه نحوه ایجاد یک نخ پوزیکس

- مطالعه نحوه استفاده از امکانات اماده نخ های 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 نا امن 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) *Need*_{*i*} <= *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 $Request_i[j] = k$ آرایه درخواست‌های فرایند P_i است. به این معنی است که فرایند P_i تعداد k تا از منبع R_j درخواست دارد. شبه کد درخواست منبع به صورت این است:

1) If $Request_i \leq Need_i$

Goto step (2); otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i \leq Available$

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:

$$Available = Available - Request_i$$

$$Allocation_i = Allocation_i + Request_i$$

$$Need_i = Need_i - Request_i$$

بانکدار:

بانکدار درخواست‌های n مشتری را برای m نوع منبع بررسی خواهد کرد. برای سادگی، فرض کنید ۶ 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 بدنهند. 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 اینمی مطرح شده در الگوریتم

بانکداران را ارضاء نماید. 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 با موقعيت، مقدار ۰ و در صورت عدم موفقیت مقدار ۱- را برمیگردانند. نخهای متعددی به صورت همروند، با استفاده از این دو تابع به دادهای مشترکشان دسترسی خواهند داشت. بنابر این، دسترسی میباشد از طریق قفل های انحصار متقابل برای پیشگیری شرایط مسابقه کنترل شود. هر دوی API های نخهای پوزیکس و ویندوز، قفل های 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 خواهد کرد:

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

آرایه available با این مقادیر مقدار اولیه می‌گیرد. می‌توانید از کد زیر استفاده کنید. روش مناسب برای مقداردهی ارایه maximum 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 تحويل داده شود.
- تمامی مراحل انجام تمرین‌ها باید مرحله به مرحله توسط تمامی دانشجویان انجام شده و تمامی آن‌ها باید پس از انجام آزمایش قادر به توضیح مراحل مختلف باشند.

مراجع مطالعه/پیوست‌ها:

5. <https://www.youtube.com/watch?v=rCssuKnHXiw>
6. <https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>
7. <https://stackoverflow.com/questions/9613934/pthread-race-condition-suspicious-behaviour>

Scheduling	موضوع:
۸	شماره آزمایش:
شبیه‌سازی الگوریتم‌های زمان‌بندی	عنوان:
	هدف:

زمان‌بندی اجرای فرآیندها جهت استفاده مناسب از منبع پردازشی سیستم‌عامل

آمادگی پیش از آزمایش:

مطالعه انواع الگوریتم‌های زمان‌بندی

شرح آزمایش:

بخش اول: برنامه‌ای به زبان C بنویسید که الگوریتم first come first serve(FCFS) را پیاده‌سازی کند. برای این کار مراحل زیر را انجام دهید.

۱. تعداد فرآیندها را از کاربر دریافت کنید.
۲. زمان سرویس‌دهی هر فرآیند را از کاربر دریافت کنید.
۳. زمان انتظار برای فرآیند اول را با صفر مقداردهی کنید.
۴. زمان انتظار سایر فرآیندها را مشخص کنید.(دقت کنید که زمان انتظار یک فرآیند برابر با زمان اجرای فرآیند قبل است. زمان اجرای یک فرآیند برابر است با مجموع زمان انتظار و زمان سرویس‌دهی).
۵. زمان اجرای فرآیندها را حساب کنید.
۶. متوسط زمان انتظار و زمان اجرا برای هر فرآیند را حساب کنید و نمایش دهید

راهنمایی: برای فرآیند می‌توانید از ساختاری مشابه ساختار زیر استفاده کنید.

```
struct process
{
    int pid;
    int bt;
    int wt,tt;
}p[10];
```

بخش دوم: برنامه‌ای به زبان C بنویسید که الگوریتم shortest job first را پیاده‌سازی کند. برای این کار مراحل زیر را انجام دهید.

۱. تعداد فرآیندها را از کاربر دریافت کنید.
۲. زمان سرویس‌دهی هر فرآیند را از کاربر دریافت کنید.
۳. زمان انتظار برای فرآیند اول را با صفر مقداردهی کنید.
۴. فرآیندها را بر اساس زمان سرویس‌دهی مرتب کنید.
۵. زمان انتظار سایر فرآیندها را مشخص کنید (دقت کنید که زمان انتظار یک فرآیند برابر با زمان اجرای فرآیند قبل است. زمان اجرای یک فرآیند برابر است با مجموع زمان انتظار و زمان سرویس‌دهی).

۶. زمان اجرای فرآیندها را حساب کنید.

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

بخش سوم: برنامه‌ای به زبان C بنویسید که الگوریتم اولویت دار (priority) را پیاده‌سازی کند. برای این کار مراحل زیر را انجام دهید.

۱. تعداد فرآیندها را از کاربر دریافت کنید.

۲. زمان سرویس‌دهی و درجه اهمیت هر فرآیند را از کاربر دریافت کنید.

۳. زمان انتظار برای فرآیند اول را با صفر مقداردهی کنید.

۴. فرآیندها را بر اساس درجه اهمیت مرتب کنید.

۵. زمان انتظار سایر فرآیندها را مشخص کنید. (دقت کنید که زمان انتظار یک فرآیند برابر با زمان اجرای فرآیند قبل است. زمان اجرای یک فرآیند برابر است با مجموع زمان انتظار و زمان سرویس‌دهی.)

۶. زمان اجرای فرآیندها را حساب کنید.

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

بخش چهارم: برنامه‌ای به زبان C بنویسید که الگوریتم Round Robin را پیاده‌سازی کند. برای این کار مراحل زیر را انجام دهید.

۱. تعداد فرآیندها را از کاربر دریافت کنید.

۲. زمان سرویس‌دهی هر فرآیند را از کاربر دریافت کنید.

۳. کوانتوم زمانی را از کاربر دریافت کنید

۴. ترتیب انجام فرآیندها را نشان دهید.

۵. متوسط زمان انتظار هر فرآیند را حساب کنید.

بخش پنجم: برای تعداد فرآیندهای به اندازه کافی بزرگ، روش‌های پیاده‌سازی شده در قبل را در قالب جدول بر اساس مشخصه‌های الگوریتم‌های زمانبند، مقایسه کنید و برای هریک دلیل بیاورید که در چه کاربردی مناسب و در چه کاربردی نامناسب است.

بخش اختیاری:

دانشجویان می‌توانند بنابر صلاح‌حدید مدرس یکی از بخش‌های فوق را بر روی هسته سیستم‌عامل اجرا کنند. در اینصورت نیازی به پیاده‌سازی بخش‌های فوق نیست.

خروجی‌های مورد انتظار آزمایش:

انتظار می‌رود دانشجویان پس از یادگیری مطالب فوق بتوانند موارد خواسته شده را انجام دهند و نتیجه را به مدرس ارائه دهند. بدین منظور لازم است نتایج بدست آمده هر بخش را در جدولی تنظیم کنید و مقایسه‌ای بین بخش‌های ۱ الی ۴ انجام دهید.

مراجع مطالعه/پیوستها:

openMP

موضوع:

۱۰

شماره آزمایش:

برنامه نویسی چند هسته ای با استفاده از openMP

عنوان:

هدف:

آشنایی با نحوه نوشتن انواع مأموریت‌های هسته و اجرای آن‌ها روی هسته

آمادگی پیش از آزمایش:

شرح آزمایش:

مقدمه :

شاید به ندرت بتوان در دنیای برنامه‌نویسی موازی رابطی خوشدست‌تر و ساده‌تر از OpenMP سرnam Multi-Processing (یافت. این رابط، انعطاف‌پذیر و ساده بوده و همچنین می‌تواند برای توسعه برنامه‌های موازی روی پلتفرم‌های مختلف به کار رود. OpenMP یک API است که می‌تواند از برنامه‌نویسی چندپردازنده‌ای به صورت حافظه اشتراکی (روی Shared-memory) C، C++، فرتون و درمعماری‌های مختلفی از جمله پلتفرم‌های ویندوز و یونیکس پشتیبانی کند. البته، تولیدکنندگان کامپایلر برای زبان‌های دیگر از جمله جاوا نیز امکان نوشتن برنامه با رابط OpenMP را فراهم کرده‌اند. در این آزمایش قصد داریم کمی با openMP آشنا شویم. ابتدا نحوه ایجاد یک پروژه با قابلیت پشتیبانی از این رابط را گام به گام دنبال کنید و سپس مراحل مسئله مورد نظر را طی کنید.

شرح مسئله :

در این آزمایش یک کد آماده در اختیار شما قرار داده می‌شود که کد یک برنامه سریال است. این کد یک مسئله ضرب دو ماتریس بسیار بزرگ است و شما باید با استفاده از این رابط آن را موازی سازی کنید.

ساختن یک پروژه با پشتیبانی OPENMP در ویندوز :

مرحله اول : ساخت پروژه در ویندوز استودیو، تنظیم پروژه، کامپایل و اجرای کد

۱. ابتدا محیط Visual Studio را اجرا کرده و از منوی File گزینه New Project را انتخاب کنید.
۲. در سمت چپ صفحه باز شده، از لیست Template، گزینه‌ی C++ Win32 را انتخاب کرده و ۳. از وسط صفحه گزینه Win32 Console Application را انتخاب کنید. مسیر و نام پروژه را مشخص کنید و کلید Next را کلیک کنید.

۴. بر روی Next کلیک کنید. Application type Console Application باید باشد. را انتخاب و Finish را کلیک کنید.

۵. در پنجره Solution Explorer بر روی پوشه Source Files کلیک کنید. سپس کلید ترکیبی Ctrl + Shift + A را فشار دهید. پیش از کلیک بر روی کلید Add اطمینان حاصل کنید که پسوند فایل شما .cpp است.
۶. کد داده شده را بخوانید و سپس در فایل قرار دهید.

۷. از نوار ابزار بالای برنامه Debug را به Release تغییر دهید.
۸. برای اجرای برنامه از کلید ترکیبی Ctrl + F5 استفاده کنید (این مرحله را پس از فعال سازی Open MP انجام دهید)

مرحله دوم : فعالسازی OpenMP و موازی سازی برنامه جهت فعال سازی OpenMP از پنجره Solution Explorer پروژه را انتخاب کنید. کلید ترکیبی Alt + Enter را فشار داده و از لیست سمت چپ از شاخه Language C\C++ آیتم Configuration را انتخاب کنید. سپس در سمت راست گزینه OpenMP Support را فعال کنید. اطمینان حاصل کنید که در بالای صفحه Platform مطابق با انتخاب مرحله اول آزمایش باشند.

ساختن یک پروژه با پشتیبانی OPENMP در لینوکس :

- ابتدا شما باید با استفاده از دستور زیر openMP را نصب کنید.

```
sudo apt install libomp-dev
```

- سپس با استفاده از دستور زیر آن را کامپایل کنید.

```
gcc -fopenmp main.c
```

شرح مسئله :

در این آزمایش یک کد آماده در اختیار شما قرار داده می‌شود که کد یک برنامه سریال است. این کد یک مسئله ضرب دو ماتریس بسیار بزرگ است و شما باید با استفاده از این رابط آن را موازی سازی کنید.

- در اولین گام زمان اجرا این کد سریال را به ازای طول ماتریس‌های داخل جدول اندازه گیری کرده و گزارش دهید.

اندازه ماتریس	۱۰۰۰۰	۱۰۰۰	۱۰۰
زمان اجرا			

- در این مرحله و با استفاده از توضیحات زیر کد برنامه را به گونه‌ای عوض کنید که به صورت موازی اجرا شود. پس از موازی سازی جدول زیر توضیحات را تکمیل نمایید.

- دستور برای ایجاد ناحیه موازی :

```
#pragma omp parallel [clause list]
{
//The code that needs to run in parallel
}
```

این دستور باعث ایجاد یک ناحیه موازی می‌شود که تعداد آن‌ها با استفاده از دستوراتی که در ادامه گفته می‌شود تنظیم می‌شود) اجرا خواهد شد.

- دستور برای تنظیم کردن تعداد نخ‌ها :

```
omp_set_num_threads(NUM_THREADS);
```

با استفاده از این دستور تعداد نخ‌هایی را که میخواهید ناحیه موازی را اجرا کنند را تنظیم میکنید. توجه کنید ممکن است درخواست شما به طور کامل انجام نشود یعنی تعداد نخ‌هایی که ناحیه موازی را اجرا میکنند از تعداد نخ‌هایی که شما درخواست داده اید کمتر باشد. علت این موضوع چیست؟

(۳) دستور برای گرفتن تعداد نخ‌های مشغول به کار در ناحیه موازی:

```
nthrds = omp_get_num_threads();
```

توجه کنید که این دستور را باید در ناحیه موازی قرار دهید.

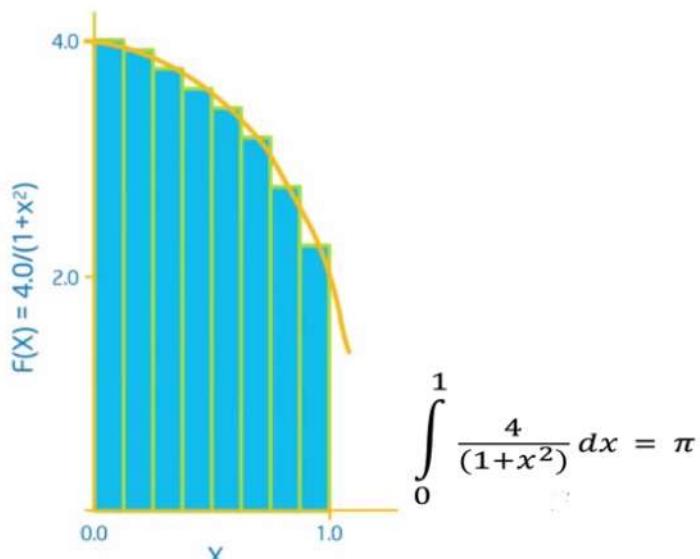
(۴) دستور برای گرفتن شناسه هر نخ داخل ناحیه موازی:

```
id = omp_get_thread_num();
```

این دستور یک عدد int که شناسه آن نخ است را برمیگرداند.

مثال:

این برنامه برای محاسبه عدد π نوشته شده است. کد اول نسخه سریال و کد دوم نسخه موازی آن است. همانطور که میدانید این عدد را با استفاده از روابط زیر میتوان محاسبه کرد.



برنامه سریال

```
static long num_steps = 100000;
double step;
```

```

void main()
{
    int i;
    double x, pi, sum = 0.0;
    step = 1.0 / (double)num_steps;
    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5)*step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
    pi = step * sum;
}

```

برنامه موازی

```

#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main() {
    int i, nthreads;
    double pi, sum[NUM_THREADS];
    step = 1.0 / (double)num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i = id, sum[id] = 0.0; i < num_steps; i = i + nthrds) {
            x = (i + 0.5)*step;
            sum[id] += 4.0 / (1.0 + x * x);
        }
    }
    for (i = 0, pi = 0.0; i < nthreads; i++) pi += step * sum[i];
}

```

۱۶	۸	۴	تعداد نخ ها
			زمان اجرا
			تسريع

```

// Example Program
// Optimizes code for maximum speed
#pragma optimize( "2", on )
#include <stdio.h>

#include <math.h>
// Adds an additional library so that timeGetTime() can be used
#pragma comment(lib, "winmm.lib")
const long int VERYBIG = 1000;
// ****
void fill(float *arr) {

```

```

        for (int i = 0; i < VERYBIG; i++) {
            arr[i] = 0.1;
        }
    }
int main(void)
{
    int i;
    long int j, k, sum;
    float total = 0;
    float A[VERYBIG], B[VERYBIG], C[VERYBIG];
    fill(&A[0]);
    fill(&B[0]);
    //
    // Output a start message
    printf("None Parallel Timings for %d iterations\n\n", VERYBIG);
    // repeat experiment several times
    for (i = 0; i < 6; i++)
    {
        // reset check sum & running total
        sum = 0;
        // Work Loop, do some work by looping VERYBIG times
        for (j = 0; j < VERYBIG; j++)
        {
            // increment check sum
            sum += 1;
            // Calculate first arithmetic series
            C[j] = A[j] * B[j];
            total = total + C[j];
        }
        printf("Total = %lf Check Sum = %ld Time = %lf\n", total, sum, ela);
        total = 0;
    }
    // return integer as required by function header
    getchar();
    return 0;
}
// *****

```

۳. با توجه به نتایج قسمت قبل تحلیل کنید چرا عدد میزان تسریع با عدد نخ های متناظر برابر نیست؟

راهنمایی :

کار اصلی در اینگونه موازی سازی ها این است که بتوانید بر اساس شناسه هر نخ، مقداری از فضای ارایه هایی را که قرار است در هم ضرب شوند به هر نخ اختصاص دهید.

خروجی های مورد انتظار آزمایش:

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

مراجع مطالعه/پیوستها:

Final project	موضوع:
۹	شماره آزمایش:
پروژه موضوعی از طرف مدرس	عنوان:
	هدف:

توسط مدرس مشخص شود.

آمادگی پیش از آزمایش:

شرح آزمایش:

توسط مدرس مشخص شود.

خروجی‌های مورد انتظار آزمایش:

توسط مدرس مشخص شود.

مراجع مطالعه/پیوست‌ها: