بسم الله الرحمن الرحیم

پروژه دوم درس مهارت های پیشرفته کار با کامپیوتر

دکتر دوستی

مهدی وجهی - ۸۱۰۱۰۱۵۵۸

فهرست

سوال ۱ - vim

۱ - توانایی کار با vim

جا به جایی

hjkl
4w
4e
gg
G
:18

نوشتن و پاک کردن

:15 | normal A 810101558
:7 | gUw
^ 2cw
dw \n 8gg \n P
R
:3 \n 3wd$
2yyG3p
2ywggP
v<count>j:normal I#
v<count>j:normal f#x
r<char>

سرچ کردن و اجرای کامند

/Host <n, N>
/Host\c <n, N>
%s/host/new_word/gci
:set hls
:terminal ip-br -c a

ذخیره و مقایسه

:w

6y:tabnew p :w PATH

:tabfisrt gg15yy :new p

vimdiff <PATH1> <PATH2>

:diffget :diffput

۲- کانفیگ vim

```
nnoremap <leader>w :w<CR>
nnoremap <leader>v :vsp<CR>
nnoremap <leader>h :sp<CR>

set relativenumber

set ignorecase

set hls

set tabstop=4

syntax on
colorscheme slate
```

۳- افزونه vim

الف- نصب پلاگین

این افزونه به ما اجازه می دهد تا به صورت درختی بین فایل ها پیمایش کنیم و فایل مورد نظر خود را آسان تر و بدون خروج از vim باز کنیم.

ب- تعریف کلید میانبر

انجام شد.

ج- نصب افزونه دلخواه

SURROUND.VIM

Surround.vim is all about "surroundings": parentheses, brackets, quotes, XML tags, and more. The plugin provides mappings to easily delete, change and add such surroundings in pairs.

It's easiest to explain with examples. Press `cs"'` inside

```
"Hello world!"
```

to change it to

```
'Hello world!'
```

Now press `cs'<q>` to change it to

```
<q>Hello world!</q>
```

To go full circle, press `cst"` to get

```
"Hello world!"
```

۴- استفاده از ماکرو

```
:reg d
Type Name Content
  c  "d   IPoet: ^[f,<80><fd>acw | Years:^[f,<80><fd>acw | Famous
Work:^[<80><fd>a0j
```

۵- سوالات تشریحی

https://g.co/gemini/share/49f033384b16

۱- اهمیت استفاده از فایل اکسپلورر

استفاده از افزونه‌ای مانند NERDTree در Vim برای مدیریت فایل‌های پروژه بسیار مهم است زیرا یک نمایش درختی بصری از ساختار دایرکتوری پروژه ارائه می‌دهد. این کار باعث می‌شود:

- پیمایش بین فایل‌ها و پوشه‌ها بسیار سریع‌تر و آسان‌تر شود.
- امکان ایجاد، حذف، تغییر نام و جابجایی فایل‌ها و دایرکتوری‌ها به راحتی و با استفاده از کلیدهای میانبر فراهم شود.

- دید کلی واضحی از ساختار پروژه به دست آید.
- سوئیچ کردن بین فایل‌ها بدون نیاز به تایپ مسیر کامل آن‌ها سریع‌تر انجام شود. در مجموع، استفاده از چنین افزونه‌ای بهره‌وری در مدیریت فایل‌ها در حین کدنویسی یا ویرایش متن را به طور قابل توجهی افزایش می‌دهد.

۲- trade-off قابلیت و عملکرد

اضافه کردن پلاگین‌ها و تنظیمات پیچیده باعث افزایش Functionality (قابلیت‌ها و کارایی) Vim می‌شود و آن را قدرتمندتر و متناسب با نیازهای شما می‌کند. با این حال، هر افزونه یا تنظیم پیچیده‌ای سربار پردازشی ایجاد می‌کند. استفاده از تعداد زیادی پلاگین، به خصوص اگر بهینه نوشته نشده باشند، یا داشتن یک .vimrc بسیار بزرگ و پیچیده می‌تواند باعث کاهش Performance (عملکرد) Vim شود، مانند کند شدن زمان راه‌اندازی، کند شدن اجرای دستورات، کند شدن هایلایت کردن نحو (syntax highlighting) و کاهش کلی سرعت واکنش‌گرایی ویرایشگر. Trade-off (مبادله) در اینجا این است که باید تعادلی بین افزودن قابلیت‌های مورد نیاز برای افزایش بهره‌وری و حفظ عملکرد مناسب Vim پیدا کنید. کاربران باید پلاگین‌ها را با دقت انتخاب کرده و .vimrc خود را مرتب و بهینه نگه دارند تا از کند شدن غیرضروری ویرایشگر جلوگیری کنند.

۳- نامه ای به نقی

سلام تقی جان،

حالت چطوره؟

می‌دونم که سال‌هاست داری با Nano کار می‌کنی و باهاش راحتی. Nano ویرایشگر ساده و دم دستی هست که برای کارهای سریع خوبه، ولی راستش رو بخوای، برای یک Sysadmin حرفه‌ای مثل تو که دائماً با فایل‌های متنی و کانفیگ‌ها سر و کار داره، ابزارهای خیلی قدرتمندتر و کارآمدتری هم وجود داره که می‌تونه سرعت و دقت کارت رو چندین برابر کنه. یکی از این ابزارها Vim هست.

شاید یادگیری اولیه Vim به نظر سخت بیاد چون مثل Notepad یا Nano نیست که همینطوری شروع کنی به تایپ کردن. Vim یک ویرایشگر **Modal** هست، یعنی حالت‌های کاری مختلفی داره (مثل حالت عادی، حالت درج، حالت بصری، حالت خط فرمان). این ایده اولیه شاید عجیب باشه، ولی همین موضوع نقطه قوت اصلی Vim هست.

چرا Vim می‌تونه انتخاب بهتری برای تو باشه؟

- **سرعت و کارایی بالا:** وقتی با حالت‌های مختلف Vim آشنا بشی، می‌تونی بدون اینکه دستت رو از روی کیبورد برداری، با سرعت نور بین متن جابجا بشی، کپی کنی، بچسبونی، حذف کنی یا حتی

تغییرات پیچیده روی بخش‌های مختلف متن اعمال کنی. دستورات Vim به شکل فوق‌العاده‌ای برای ویرایش بهینه متن طراحی شدن.

- **قدرت و انعطاف‌پذیری بی‌نظیر:** Vim به شدت قابل شخصی‌سازی هست. می‌تونی `.vimrc` خودت رو طوری تنظیم کنی که دقیقاً بر اساس نیازهای تو عمل کنه. پشتیبانی از صدها زبان برنامه‌نویسی و فایل کانفیگ با هایلایت نحو، قابلیت استفاده از افزونه‌های قدرتمند برای تکمیل خودکار کد، مدیریت پروژه (مثل همون NERDTree که بالاتر گفتم)، و خیلی کارهای دیگه، Vim رو تبدیل به یک نیروگاه ویرایش متن می‌کنه. کارهایی که با Nano خیلی وقت‌گیر یا غیرممکن هستن، در Vim به راحتی و با چند کلید انجام میشن.
- **حضور همه‌جا:** Vim (یا نسخه قدیمی‌ترش Vi) روی تقریباً تمام سیستم‌های مبتنی بر یونیکس و لینوکس به صورت پیش‌فرض وجود داره. این یعنی هرجا که لاگین کنی، حتی در محیط‌های ریکاوری یا سیستم‌های حداقل، یک ویرایشگر آشنا و قدرتمند در دسترس داری. این برای یک Sysadmin خیلی حیاتیه.
- **ماکروها و اتوماسیون:** Vim بهت اجازه می‌ده دنباله‌ای از کارهایی که انجام میدی رو ضبط کنی (ماکرو) و بعد هر چند بار که خواستی تکرار کنی. این برای انجام کارهای تکراری روی فایل‌های بزرگ یا مجموعه‌ای از فایل‌ها فوق‌العاده مفیده و ساعت‌ها وقتت رو ذخیره می‌کنه.

قبول دارم که منحنی یادگیری Vim از Nano بالاتره، ولی وقت کمی که برای یادگیری مفاهیم اولیه Vim میذاری، در طولانی مدت صدها ساعت در وقتت صرفه‌جویی می‌کنه. انگار که داری یک ابزار فوق پیشرفته یاد می‌گیری که مخصوص کار تو ساخته شده.

پیشنهاد می‌کنم برای چند روز هم که شده، Vim رو امتحان کنی و سعی کنی کارهای روزمره‌ات رو باهاش انجام بدی. آموزش‌های آنلاین و منابع زیادی برای شروع وجود داره. مطمئنم بعد از مدتی استفاده، تفاوت فاحش رو احساس می‌کنی و دیگه دلت نمی‌خواد برگردی به Nano!

حالا تصمیم با خودته، ولی حیف نیست از این ابزار قدرتمند بی‌بهره بمونی؟

۴- مزایای ماکرو

ماکروها در Vim قابلیتی بسیار قدرتمند برای اتوماسیون کارهای تکراری در ویرایش متن هستند. مزایای استفاده از آن‌ها عبارتند از:

- **اتوماسیون:** ضبط یک دنباله از دستورات و اجرای مجدد آن چندین بار به طور خودکار، نیاز به تکرار دستی را از بین می‌برد.
- **افزایش سرعت:** کارهایی که نیاز به اعمال مجموعه‌ای یکسان از تغییرات در قسمت‌های مختلف فایل دارند (مانند فرمت‌بندی داده‌ها، تغییر ساختار کد)، با استفاده از ماکرو به سرعت انجام می‌شوند.

- **دقت و یکپارچگی:** چون یک مجموعه دستور دقیق اجرا می‌شود، خطاهای ناشی از تکرار دستی کاهش می‌یابند و تغییرات به صورت یکپارچه اعمال می‌شوند.
- **انعطاف‌پذیری:** ماکروها را می‌توان به راحتی ضبط کرد و روی خطوط یا انتخاب‌های مختلف متن اجرا کرد.

۵- مشکلات ماکرو

مشکلات رایجی که ممکن است هنگام ضبط یا اجرای مجدد ماکروها پیش بیایند شامل موارد زیر هستند:

- **ضبط اشتباه:** اشتباهی شامل کردن کلیدهای ناخواسته یا دستورات غلط در حین ضبط.
- **وابستگی به موقعیت مکان‌نما (Cursor):** بسیاری از ماکروها به موقعیت شروع مکان‌نما نسبت به متنی که روی آن عمل می‌کنند وابسته هستند. اجرای ماکرو از یک نقطه شروع اشتباه، نتیجه غلطی به دنبال خواهد داشت.
- **تغییر ساختار متن:** اگر ساختار متن بین زمان ضبط و زمان اجرای ماکرو تغییر کند، ممکن است دستورات ضبط شده نتوانند الگوهای مورد انتظار را پیدا کنند یا به موقعیت‌های صحیح بروند، که باعث خرابی یا عملکرد غیرمنتظره ماکرو می‌شود.
- **قطع شدن ناخواسته ضبط:** مثلاً فشار دادن اشتباهی کلید q برای بار دوم قبل از پایان کار.
- **دستورات حالت‌دار (Modal Commands):** فراموش کردن اینکه چه دستوری در چه حالتی اجرا می‌شود (مثلاً اشتباه گرفتن دستورات Normal mode با Insert mode).

در حل این سوال من به مشکلی نخوردم.

۶- ویرایشگر های با/بدون حالت

ویرایشگرهای متن به طور کلی به دو دسته اصلی تقسیم می‌شوند:

- **Editorهای Modeless (بدون حالت):** در این ویرایشگرها، شما همیشه در یک حالت کاری هستید. معمولاً تایپ کردن کاراکترها باعث درج مستقیم آن‌ها در متن می‌شود و برای انجام دستوراتی مانند ذخیره، کپی، چسباندن یا جستجو باید از ترکیبات کلیدهای کنترلی (مثل Ctrl+S, Ctrl+C, Ctrl+V) یا منوها و نوار ابزار استفاده کنید. مثال‌ها: Notepad, VS Code, Sublime Text, Nano, Microsoft Word.
- **Editorهای Modal (حالت‌دار):** این ویرایشگرها چندین "حالت" کاری مجزا دارند که هر کدام برای انجام کارهای خاصی طراحی شده‌اند. مثلا vim از این دسته است و چند حالت نرمال و insert را دارد.

۷- مد های vim

- **Normal Mode (حالت عادی):** حالت پیش‌فرض که برای جابجایی در متن، حذف، کپی، چسباندن، و اجرای دستورات ویرایشی قدرتمند با فشردن یک یا ترکیبی از چند کلید استفاده می‌شود. در این حالت تایپ کردن مستقیماً متن را درج نمی‌کند.
- **Insert Mode (حالت درج):** حالتی که برای تایپ مستقیم متن استفاده می‌شود. برای ورود به این حالت از Normal mode (با کلیدهایی مانند i, a, o) و برای خروج از آن (معمولاً با Esc) به Normal mode برمی‌گردید.
- **Visual Mode (حالت بصری):** برای انتخاب بلوک‌های متن به صورت بصری.
- **Command-line Mode (حالت خط فرمان):** برای تایپ دستورات پیشرفته که با کاراکترهای خاصی مانند :, / یا ? شروع می‌شوند.

سوال ۲- vcs

۱- بررسی پروژه

```
@mvajhi ➜ /workspaces/git (master) $ git log --pretty=oneline --all |
wc -l
80218

@mvajhi ➜ /workspaces/git (master) $ git log --author="torvalds"
--since="5 years ago" --pretty=oneline | wc -l
1

@mvajhi ➜ /workspaces/git (master) $ git log --author="torvalds"
--before="16 years ago" --pretty=oneline | wc -l
1045

@mvajhi ➜ /workspaces/linux (master) $ git log --pretty=oneline --
kernel/pid_namespace.c | wc -l
103

@mvajhi ➜ /workspaces/linux (master) $ git log --diff-filter=A --all --
kernel/pid_namespace.c
commit 74bd59bb39eb08b4379e2590c5f160748d83f812
Author: Pavel Emelyanov <xemul@openvz.org>
Date:   Fri Feb 8 04:18:24 2008 -0800
```

۲- کانفیگ

```
git config --local user.name "Mahdi Vajhi"
git config --local user.email mvajhimv@gmail.com
```

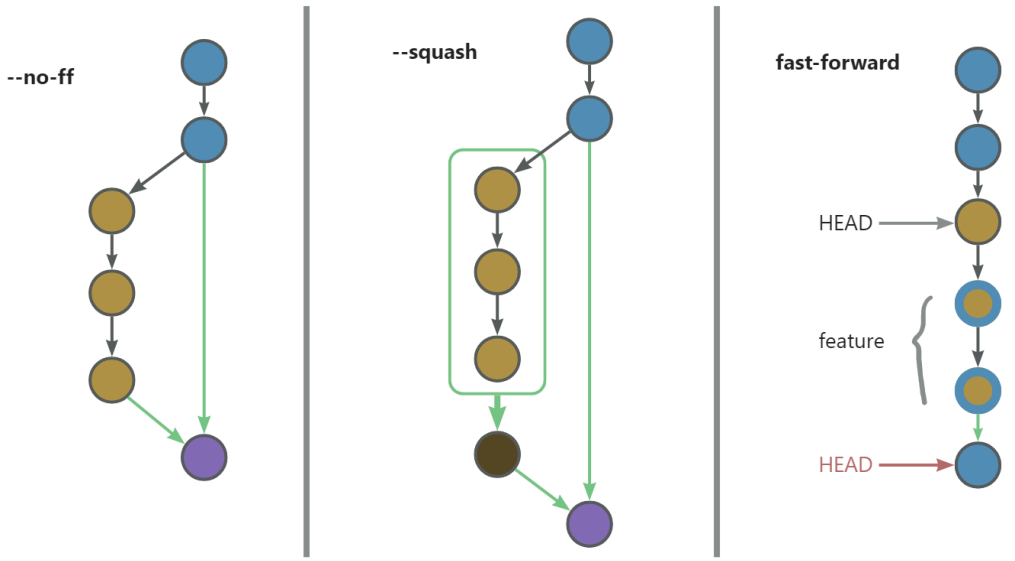
لوکال فقط شامل همون پروژه میشه، گلوبال شامل اون یوزر با همه مخازنش میشه و سیستم برای کل سیستم هست.

```
mvajhi@mahdi-laptop essential-computing-skills/CA2 (main) » git config
--global alias.lg "log --oneline --all --graph"
mvajhi@mahdi-laptop essential-computing-skills/CA2 (main) » git config
--global alias.bis "bisect"
mvajhi@mahdi-laptop essential-computing-skills/CA2 (main) » git lg
mvajhi@mahdi-laptop essential-computing-skills/CA2 (main) » git bis
fatal: need a command
```

```
usage: git bisect start [--term-(new|bad)=<term>
--term-(old|good)=<term>]     [--no-checkout] [--first-parent] [<bad>
[<good>...]] [--] [<pathspec>...]
   or: git bisect (good|bad) [<rev>...]
   or: git bisect terms [--term-good | --term-bad]
   or: git bisect skip [(<rev>|<range>)...]
   or: git bisect next
   or: git bisect reset [<commit>]
   or: git bisect visualize
   or: git bisect replay <logfile>
   or: git bisect log
   or: git bisect run <cmd> [<arg>...]
```

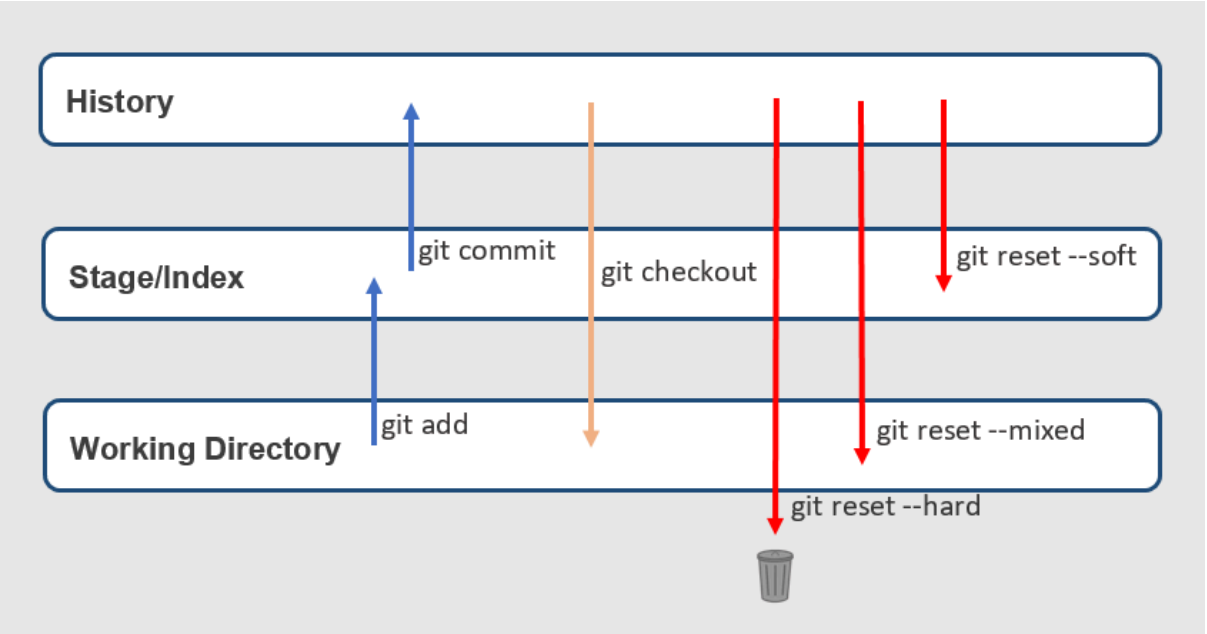
۳- سناریو

مشخصا مرج از نوع no-ff هست در تصویر زیر ۳ مدل مرج رو مشاهده می کنید.

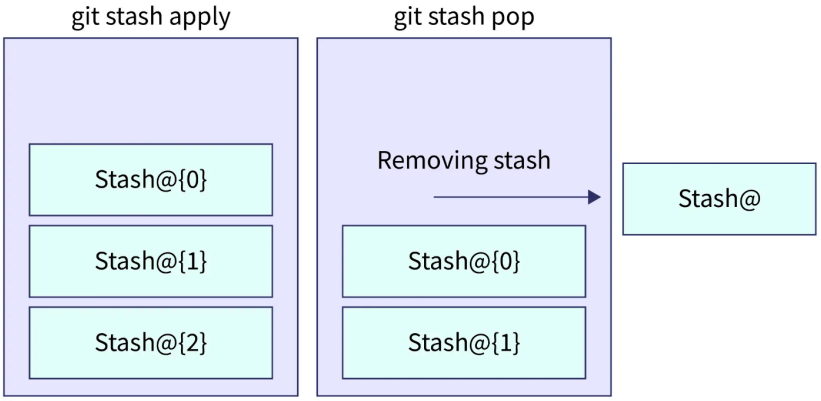


در ff سابقه شاخه در شاخه مرج شده می آید. در اسکواش کل شاخه در قالب یک کامیت قرار می گیرد و خود شاخه حذف می شود و تنها آن کامیت می ماند. no-ff با یک کامیت تغییرات را در شاخه ی مرج شده قرار می دهد. البته حالت های دیگری مثل rebase هم هست اما موارد اصلی این ها بودند.

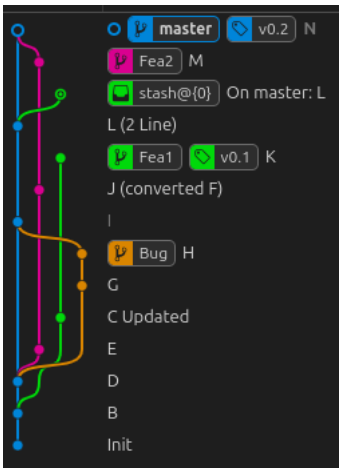
در این تصویر به خوبی تفاوت reset های مختلف را مشاهده می کنید. ما در این جا از ریست هارد استفاده کردیم.



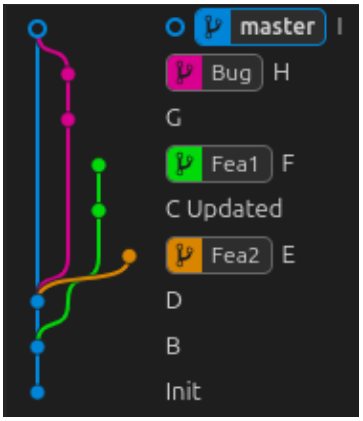
تفاوت git stash apply & pop در تصویر زیر مشخص است:



نتیجه گیت گرافی به شکل زیر است:



همچنین قبل از انتقال کامیت F گراف به شکل زیر است:



کامند ها در پروژه قرار دارد.

hg -۴

گراف لاگ به این صورت است:

Symbols explanation

	Graph edges
|	Regular connection (vertical)
\	Regular connection (diagonal left)
/	Regular connection (diagonal right)
-	Regular connection (horizontal, used with +)
+	Graph branching point
:	There are nodes between the nodes on this line, but they've been omitted
~	The graph was cut off to exclude the node on the other end of this line

	Graph nodes
@	The commit you're currently on
_	Branch was closed
x	Obsolete commit (ChangesetEvolution)
*	Unstable commit (ChangesetEvolution)
o	Regular commit

```
|\ \ \
| _ | |  104 Branch was closed
| | | |
| @ | |  103 The commit that you're currently on
| | | |
| | | o  102 Regular commit, child of 100 (not 101)
| | | |
|/ / /
o---+    101 Merge of 99 and 100
| | |
| | o    100
| | |
o | |    99
| | |
| * |    98 Unstable
| | |
| x |    97 Obsolete
```

هم چنین علاوه بر دستور لاگ صورت پروژه از این دستور با تنظیم دلخواه هم می توان استفاده کرد.

```
hg log -G -T '{node|short} {tags}{branches} {desc|firstline}\n'
```

در نهایت درختی به شکل زیر تشکیل می شود:

```
@     tip E merge F2
|\
| o   D merge F1
| |\
o | |   Fea2 F2B
| | |
o---+   Fea2 F2A
 / /
| o   C
| |
o |   Fea1 F1B
| |
o |   Fea1 F1A
|/
o     B
|
o     A
```

کامند ها در پروژه قرار دارد.

۵- سوالات تشریحی

https://github.com/copilot/share/02031086-0a80-80a3-b910-044484294866

https://chatgpt.com/share/681d2b31-909c-8001-967b-b245c1feff80

۱- blame, bisect

- git bisect: این ابزار برای پیدا کردن کامیتی که باعث ایجاد یک باگ شده است، استفاده می‌شود. با تقسیم تاریخچه گیت به دو بخش (خوب و بد)، می‌توانید کامیت مشکل‌ساز را با جستجوی باینری پیدا کنید.
- git blame: این ابزار برای مشاهده تغییرات خط به خط و مشخص کردن نویسنده هر تغییر استفاده می‌شود.

سناریو زیر را فرض کنید:

```
* 024b0c5 (master) Added C
* 31d6847 Added B
* 9ed55f1 Bug here
* 4a03c2b Ok 2
* b5a18a8 Ok commit
```

حال شروع می کنیم کامیت فعلی را بد و اولی را خوب قرار می دهیم.

```
vajhi@mahdi-laptop /tmp/tg (master) » git bisect start
status: waiting for both good and bad commits
mvajhi@mahdi-laptop /tmp/tg (master) » git lg
mvajhi@mahdi-laptop /tmp/tg (master) » git bis good b5a18a8
status: waiting for bad commit, 1 good commit known
mvajhi@mahdi-laptop /tmp/tg (master) » git bis bad
Bisecting: 1 revision left to test after this (roughly 1 step)
[9ed55f15869bc406f3b0ef6fc14bc9eec4a2ed07] Bug here
mvajhi@mahdi-laptop /tmp/tg » git bis visualize
mvajhi@mahdi-laptop /tmp/tg » git bis bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[4a03c2b982a0c91ed7e563527a80bbd19cabb961] Ok 2
mvajhi@mahdi-laptop /tmp/tg » git bis visualize
mvajhi@mahdi-laptop /tmp/tg » git bis good
9ed55f15869bc406f3b0ef6fc14bc9eec4a2ed07 is the first bad commit
commit 9ed55f15869bc406f3b0ef6fc14bc9eec4a2ed07
Author: mvajhi <mvajhimv@gmail.com>
Date:   Fri May 9 01:58:31 2025 +0330

      Bug here

 s.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

همچنین blame هم می تواند به ما کمک کند:

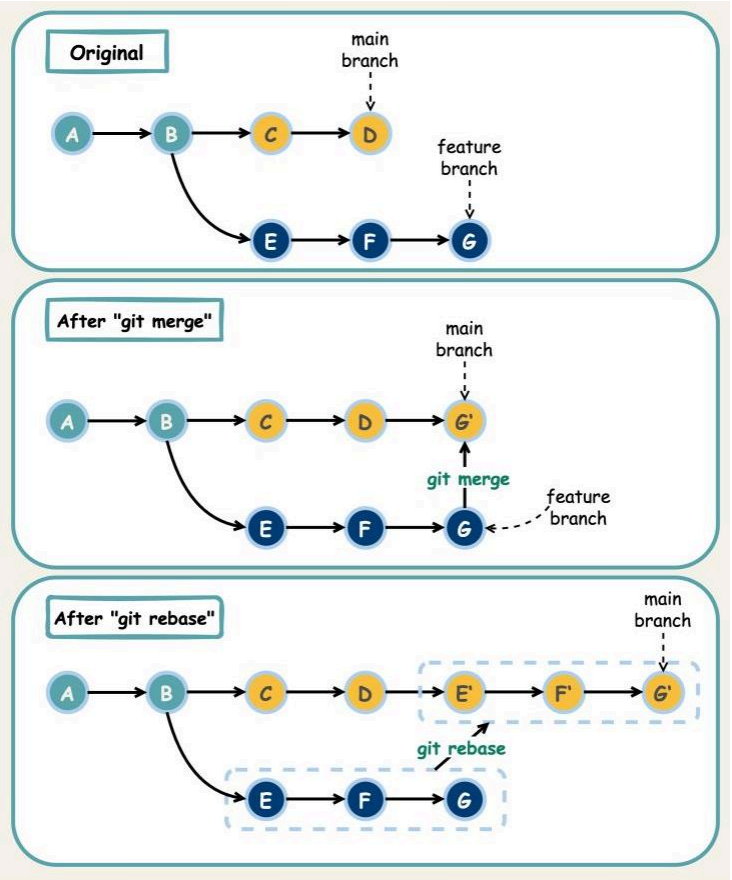
```
^b5a18a8 (mvajhi 2025-05-09 01:57:12 +0330 1) def sum(a, b):
9ed55f15 (mvajhi 2025-05-09 01:58:31 +0330 2)   return a - b
^b5a18a8 (mvajhi 2025-05-09 01:57:12 +0330 3)
^b5a18a8 (mvajhi 2025-05-09 01:57:12 +0330 4) def test_sum():
^b5a18a8 (mvajhi 2025-05-09 01:57:12 +0330 5)   assert sum(2, 2) == 4
^b5a18a8 (mvajhi 2025-05-09 01:57:12 +0330 6)   assert sum(3, 5) == 8
^b5a18a8 (mvajhi 2025-05-09 01:57:12 +0330 7)
^b5a18a8 (mvajhi 2025-05-09 01:57:12 +0330 8) test_sum()
```

۲- blameless postmortem

یک روش برای بررسی و تحلیل مشکلات یا خرابی‌ها بدون سرزنش افراد. در این فرآیند، تمرکز بر روی یادگیری از خطاها و بهبود فرآیندها است، نه پیدا کردن مقصر.در این روش تیم‌ها بعد از رخ دادن یک مشکل یا خرابی، جلسه‌ای برگزار کرده و به بررسی علت‌های اصلی مشکل می‌پردازند. هدف این است که با شناسایی

دلایل واقعی مشکل، از تکرار آن جلوگیری شود. این روش به ایجاد محیطی امن برای بیان اشتباهات کمک می‌کند.

۳- rebase



در rebase کردن تاریخچه گیت بازنویسی می شود. این موضوع می تواند به تمیزتر شدن تاریخچه کمک کند اما می تواند باعث ابهام برای سایر افراد تیم شود همچنین امکان ایجاد خرابکاری را هم می تواند فراهم کند. در مرج تاریخچه بازنویسی نمی شود. استفاده از آن در پروژه های تیمی بزرگ به هیچ عنوان استفاده نمیشه اگر هم قرار استفاده بشه حتما باید مسئول تیم و مسئول گیت و دواپس این کار را انجام دهند. دلایل آن در بالا توضیح داده شد.

۴- fetch, pull

- git fetch: تغییرات جدید را از مخزن ریموت دریافت می‌کند، اما آنها را به شاخه کاری شما اعمال نمی‌کند. این دستور برای بررسی تغییرات قبل از اعمال مفید است.
- git pull: ترکیبی از git fetch و git merge است. تغییرات را دریافت کرده و به طور خودکار به شاخه کاری شما اعمال می‌کند.

۵- تفاوت گیت و hg

ویژگی	Git	Mercurial (hg)
مدل توسعه	توزیع‌شده (Distributed)	توزیع‌شده (Distributed)
رابط کاربری (CLI)	پیچیده‌تر، اما انعطاف‌پذیرتر	ساده‌تر و کاربرپسندتر
ساختار داخلی	اشیاء (objects) و گراف مبتنی بر SHA-1	تغییرات مبتنی بر فایل و ساده‌تر
سیستم branch	قدرتمند و سبک (lightweight)	ساده‌تر ولی محدودتر
پشتیبانی از rename	از طریق تشخیص خودکار (heuristic)	از طریق دنبال‌کردن تغییرات فایل
کارایی	سریع‌تر در مخازن بزرگ	کمی کندتر در مخازن بزرگ
پشتیبانی جامعه	بسیار گسترده، پروژه‌های بزرگ مثل لینوکس	کمتر از Git، ولی هنوز فعال
میزان محبوبیت	بسیار محبوب (استاندارد صنعتی)	کمتر محبوب
مستندسازی و منابع	زیاد و گسترده	محدودتر ولی با کیفیت
سازگاری با ابزارها	به‌خوبی با GitHub، GitLab و CI/CD ها ادغام شده	سازگاری کمتر با ابزارهای مدرن
سیستم عامل‌ها	چندسکویی (Cross-platform)	چندسکویی (Cross-platform)

به خود گیت هاب که نمیشه وصل شد (البته راه هایی مثل تبدیل hg به گیت موقع پوش و تبدیل برعکس موقع پول هست که معقول نیست اما ممکن است). اما موارد مشابهی مثل sourceforge که از hg هم پشتیبانی می کند.

سوال ۳- مدیریت سیگنال های لینوکس

https://github.com/copilot/share/88035204-0b80-8c05-9153-060480e96926

۱- کشتن پردازه

با ctrl+z برنامه رو suspend می کنیم و سپس پراسس مربوطه را kill می کنیم. (pid بعد از suspend شدن نشان داده می شود.)

۲- هندل کردن همه سیگنال ها

- SIGINT (Ctrl+C): برنامه را به طور منظم خاتمه می‌دهد اگر هندل نشده باشد.
- SIGTSTP (Ctrl+Z): برنامه را متوقف می‌کند، اما خاتمه نمی‌دهد.
- SIGQUIT (Ctrl+\): برنامه را خاتمه می‌دهد و یک core dump برای دیباگ ایجاد می‌کند.

راهی برای گرفتن اون ترمینال در همون ترمینال نیست باید در ترمینالی دیگر آن پردازه را ازبین برد و به این صورت ترمینال را پس گرفت.

۳- sigaction

ویژگی	`signal`	`sigaction`
سادگی استفاده	ساده‌تر و مناسب برای برنامه‌های کوچک	پیچیده‌تر ولی مناسب برای برنامه‌های پیشرفته
قابلیت کنترل رفتار سیگنال‌ها	محدود	کنترل دقیق‌تر و انعطاف‌پذیرتر
بازگشت به توابع مسدودکننده	به صورت پیش‌فرض از توابع مسدودکننده باز نمی‌گردد و ممکن است رفتار غیرمنتظره داشته باشد	با استفاده از فلگ `SA_RESTART` عملیات مسدودکننده ادامه پیدا می‌کنند
مدیریت ایمنی در برابر سیگنال‌ها	احتمال بروز مشکلاتی مانند شرایط مسابقه (Race Condition) وجود دارد	امکان بلاک کردن سیگنال‌های دیگر هنگام اجرای هندلر برای جلوگیری از شرایط مسابقه
پشتیبانی از ویژگی‌های پیشرفته	فقط هندلر ساده را پشتیبانی می‌کند	پشتیبانی از هندلر پیشرفته با `SA_SIGINFO` برای اطلاعات بیشتر درباره سیگنال
سازگاری با سیستم‌های مدرن	ممکن است در برخی سیستم‌ها رفتار غیرقابل پیش‌بینی داشته باشد	قابل اعتمادتر و استانداردتر
بلاک کردن سیگنال‌ها در هندلر	به طور پیش‌فرض بلاک نمی‌کند	امکان بلاک کردن سیگنال‌های دیگر هنگام اجرای هندلر
امنیت	مستعد خطا و آسیب‌پذیری‌هایی مثل شرایط مسابقه است	امن‌تر و قابل اطمینان‌تر
کاربرد در پروژه‌های بزرگ	نامناسب	بسیار مناسب
فلگ‌ها (Flags)	فاقد فلگ برای تنظیمات اضافه	پشتیبانی از فلگ‌هایی مثل `SA_RESTART`، `SA_SIGINFO` و غیره

سوال ۴- برنامه نویسی کرنل

نوشتن ماژول

کد زیر نمونه ای بسیار ساده از یک ماژول لینوکس است.

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
static int __init hello_world_init(void) {
                printk(KERN_INFO "Hello, Kernel!\n");
                return 0;
}
static void __exit hello_world_exit(void) {
                printk(KERN_INFO "Goodbye, Kernel!\n");
}
module_init(hello_world_init);
module_exit(hello_world_exit);
MODULE_LICENSE("BSD");
MODULE_AUTHOR("Mahdi Vajhi");
MODULE_DESCRIPTION("A simple hello world driver");
MODULE_VERSION("1.0.0v");
```

اضافه کردن ماژول

ابتدا make می کنیم و سپس با دستور مربوطه اضافه و حذف می کنیم سپس لاگ های سیستم را بررسی می کنیم:

```
May 09 12:26:30 mahdi-laptop kernel: [UFW BLOCK] IN=wlo1 OUT=
MAC=01:00:5e:00:00:01:00:31:9d:18:ef:ca:08:00 SRC=0.0.0.0 DST=224.0.0.1
LEN=32 TOS=0x00 PREC=0xC0 TTL=1 ID=0 DF PROTO=2
May 09 12:26:49 mahdi-laptop sudo[786461]:   mvajhi : TTY=pts/2 ;
PWD=/home/mvajhi/code/essential-computing-skills/CA2/P4 ; USER=root ;
COMMAND=/usr/sbin/insmod hello_kernel.ko
May 09 12:26:49 mahdi-laptop kernel: Hello, Kernel!
May 09 12:26:52 mahdi-laptop sudo[786473]:   mvajhi : TTY=pts/2 ;
PWD=/home/mvajhi/code/essential-computing-skills/CA2/P4 ; USER=root ;
COMMAND=/usr/sbin/rmmod hello_kernel.ko
May 09 12:26:52 mahdi-laptop kernel: Goodbye, Kernel!
```

سوالات تشریحی

https://github.com/copilot/share/40620314-02a0-8c85-b951-9405a0696037

۱- تفاوت میان Microkernel و Monolithic kernel

1. Microkernel: در این نوع طراحی کرنل، تنها سرویس‌های اصلی و حیاتی سیستم‌عامل (مانند مدیریت حافظه و پردازش‌ها) در کرنل اجرا می‌شوند، و سایر سرویس‌ها (مانند درایورهای دستگاه و سیستم فایل) در فضای کاربر اجرا می‌شوند. این طراحی باعث افزایش پایداری سیستم می‌شود، زیرا خرابی یک سرویس تأثیری بر کل سیستم ندارد. اما به دلیل تعداد بیشتر تبادل پیام‌ها بین اجزای سیستم، ممکن است کارایی کاهش یابد.
2. Monolithic Kernel: در این طراحی، تمام سرویس‌های سیستم‌عامل در کرنل اجرا می‌شوند. این باعث افزایش کارایی می‌شود زیرا همه اجزا در یک فضای حافظه مشترک عمل می‌کنند. اما خرابی یک سرویس می‌تواند کل سیستم را تحت تاثیر قرار دهد.

۲- Kernel Panic

Kernel Panic زمانی رخ می‌دهد که سیستم‌عامل با خطای غیرقابل بازیابی مواجه شود که نمی‌تواند آن را مدیریت کند. این معمولاً به دلیل مشکلات درایورها، خرابی سخت‌افزار، یا خطاهای نرم‌افزاری در کرنل اتفاق می‌افتد. سیستم معمولاً در این شرایط متوقف شده و پیامی به کاربر نمایش می‌دهد که دلیل خطا را توضیح می‌دهد. برای مدیریت این وضعیت، کرنل معمولاً از مکانیزم‌های بازیابی خودکار (مانند reboot) یا ابزارهای خطایابی (مانند crash dump) استفاده می‌کند.

۳- Debug کردن یک ماژول کرنل یا درایور

برای دیباگ کردن، ابزارهای متعددی وجود دارد:

- GDB: ابزار دیباگر معروف که می‌تواند برای دیباگ کرنل با استفاده از kernel debugging stub و یا QEMU استفاده شود.
- Ftrace: ابزار داخلی لینوکس برای ردیابی رفتار کرنل.
- Crash Utility: برای تحلیل crash dump و بررسی وضعیت کرنل.
- KDB/KKGDB: دیباگرهای تعبیه‌شده برای کرنل لینوکس.

برای مثال، با استفاده از GDB، می‌توانید کرنل را روی یک ماشین مجازی (با استفاده از QEMU) بوت کنید و با دستورات GDB به تحلیل و دیباگ بپردازید. مانند آنچه در درس سیستم عامل انجام دادیم.

سوال ۵- CHAR DEVICE DRIVER

Copilot agent: ECS

https://github.com/copilot/share/804b4206-43a0-8485-a912-940ca4a96975

نوشتن ماژول

در ابتدا و بعد از تعریف کتابخانه ها اطلاعات ماژول رو می نویسیم.

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Mahdi Vajhi");
MODULE_DESCRIPTION("Character-by-Character Virtual LIFO Driver");
MODULE_VERSION("1.0v");
```

در این بخش، تنظیمات اولیه ماژول و ساختارهای کلیدی دستگاه تعریف شده‌اند. نام دستگاه‌ها مانند lifo_read و lifo_write مشخص شده‌اند که به ترتیب برای خواندن و نوشتن داده‌ها استفاده می‌شوند. این نام‌ها بعدها برای ایجاد گره‌های دستگاه در مسیر /dev به کار خواهند رفت. کلاس دستگاه با نام lifo تعریف شده است که به سازمان‌دهی و مدیریت دستگاه‌ها در مسیر /sys/class کمک می‌کند. همچنین، یک محدودیت به نام LIFO_MAX_SIZE برای تعیین حداکثر اندازه بافر LIFO (در اینجا 1 مگابایت) تعریف شده است.

متغیرهای lifo_read_device، lifo_class، dev_number و lifo_write_device به صورت static تعریف شده‌اند تا فقط در محدوده فایل فعلی در دسترس باشند و از تداخل احتمالی با دیگر بخش‌های کرنل یا ماژول‌های دیگر جلوگیری شود. این متغیرها برای مدیریت شماره‌های دستگاه (major و minor)، کلاس دستگاه و گره‌های مربوط به خواندن و نوشتن به کار می‌روند. همچنین، دو ساختار cdev با نام‌های lifo_read_cdev و lifo_write_cdev برای ثبت دستگاه‌های کاراکتری استفاده می‌شوند.

ساختار داده struct lifo_char نماینده‌ی یک گره در LIFO است و شامل یک کاراکتر داده (char data) و یک گره لیست پیوندی (struct list_head list) می‌باشد. از این ساختار برای ذخیره داده‌ها در لیست پیوندی دوطرفه استفاده می‌شود. این طراحی به ما اجازه می‌دهد تا داده‌ها را به راحتی به لیست اضافه یا از آن حذف کنیم و رفتار LIFO را پیاده‌سازی کنیم.

```
#define DEVICE_NAME_READ "lifo_read"
#define DEVICE_NAME_WRITE "lifo_write"
#define CLASS_NAME "lifo"
#define LIFO_MAX_SIZE (1024 * 1024)  // 1 MB buffer size

// Device numbers and structures
static dev_t dev_number;
static struct class *lifo_class = NULL;
static struct device *lifo_read_device = NULL;
```

```
static struct device *lifo_write_device = NULL;
static struct cdev lifo_read_cdev, lifo_write_cdev;

// LIFO character structure
struct lifo_char {
      char data;
      struct list_head list;
};
```

در ادامه داریم:

```
// LIFO stack and synchronization
static LIST_HEAD(lifo_stack);
static size_t total_size = 0;
static DEFINE_MUTEX(lifo_mutex);
static DECLARE_WAIT_QUEUE_HEAD(lifo_read_queue);
```

این بخش، ساختار داده LIFO و ابزارهای همگام‌سازی تعریف شده‌اند.(LIST_HEAD(lifo_stack یک لیست پیوندی دوطرفه (doubly linked list) ایجاد می‌کند که به عنوان پشته (stack) عمل می‌کند. این لیست با استفاده از ساختار struct list_head از کرنل لینوکس پیاده‌سازی شده است. داده‌هایی که به LIFO اضافه می‌شوند به ابتدای این لیست اضافه می‌شوند و داده‌هایی که خوانده می‌شوند نیز از ابتدای لیست حذف می‌شوند.

متغیر total_size برای نگهداری اندازه کل داده‌های موجود در LIFO استفاده می‌شود. این متغیر نشان می‌دهد که چه مقدار داده (بر حسب تعداد کاراکتر) در حال حاضر در LIFO ذخیره شده است.

برای جلوگیری از دسترسی همزمان به LIFO و حفظ انسجام داده‌ها، یک Mutex با نام lifo_mutex تعریف شده است (با استفاده از ماکروی DEFINE_MUTEX). این Mutex تضمین می‌کند که عملیات خواندن و نوشتن به LIFO به صورت ایمن و غیرهمزمان انجام می‌شود.

همچنین، صف انتظار lifo_read_queue با استفاده از ماکروی DECLARE_WAIT_QUEUE_HEAD تعریف شده است. این صف برای مدیریت فرآیندهایی که منتظر داده در LIFO هستند استفاده می‌شود. اگر فرآیندی بخواهد داده‌ای از LIFO بخواند ولی LIFO خالی باشد، این فرآیند در صف انتظار قرار می‌گیرد تا زمانی که داده‌ای به LIFO اضافه شود.

در این قسمت توابع و ساختار داده های مربوط به فایل را به تعریف می کنیم:

```
// Device operations prototypes
static int lifo_open(struct inode *inode, struct file *file);
static int lifo_release(struct inode *inode, struct file *file);
static ssize_t lifo_read(struct file *file, char __user *buf, size_t
count, loff_t *offset);
static ssize_t lifo_write(struct file *file, const char __user *buf,
size_t count, loff_t *offset);
```

```
// File operations structures
static const struct file_operations lifo_read_fops = {
	.owner = THIS_MODULE,
	.open = lifo_open,
	.release = lifo_release,
	.read = lifo_read,
};

static const struct file_operations lifo_write_fops = {
	.owner = THIS_MODULE,
	.open = lifo_open,
	.release = lifo_release,
	.write = lifo_write,
};
```

در این بخش، عملیات اصلی دستگاه‌های LIFO تعریف شده‌اند. این عملیات برای مدیریت رفتار دستگاه‌های کاراکتری (character devices) ضروری هستند و شامل مواردی مانند باز کردن، بستن، خواندن و نوشتن می‌باشند.

- lifo_open و lifo_release: این توابع به ترتیب هنگام باز کردن و بستن دستگاه توسط فرآیندها فراخوانی می‌شوند. این توابع مسئول مدیریت دسترسی به دستگاه و به‌روزرسانی شمارنده مرجع ماژول هستند.
- lifo_read و lifo_write: این توابع به ترتیب برای خواندن داده‌ها از LIFO و نوشتن داده‌ها به LIFO استفاده می‌شوند. این توابع تعامل میان فضای کاربر (user space) و فضای کرنل (kernel space) را مدیریت می‌کنند.
- ساختارهای file_operations: این ساختارها رفتار دستگاه‌های خواندن (lifo_read) و نوشتن (lifo_write) را مشخص می‌کنند. هر ساختار شامل اشاره‌گرهایی به توابع مربوطه است که عملیات دستگاه را مدیریت می‌کنند. برای مثال، ساختار lifo_read_fops تابع مربوط به read را به lifo_read متصل می‌کند و همچنین توابع دیگری مانند open و release را تعریف می‌کند.

تمام این توابع و ساختارها به صورت static تعریف شده‌اند، زیرا فقط در این فایل مورد استفاده قرار می‌گیرند و نیازی به دسترسی از فایل‌های دیگر ندارند. این کار باعث کپسوله‌سازی و جلوگیری از تداخل با دیگر بخش‌های کرنل یا ماژول‌های دیگر می‌شود.

در ادامه دوتابع برای بستن و باز شدن فایل تعریف می کنیم که این توابع در هنگام این عملیات ها صدا می شوند:

```
static int lifo_open(struct inode *inode, struct file *file)
{
	try_module_get(THIS_MODULE);
	return 0;
}
```

```
static int lifo_release(struct inode *inode, struct file *file)
{
	module_put(THIS_MODULE);
	return 0;
}
```

lifo_open: وقتی دستگاه باز می‌شود، تابع try_module_get() فراخوانی می‌شود. این تابع باعث می‌شود شمارنده مرجع ماژول افزایش یابد و از حذف ماژول در حالی که در حال استفاده است جلوگیری کند.
lifo_release: زمانی که دستگاه بسته می‌شود، تابع module_put() فراخوانی می‌شود. این تابع شمارنده مرجع ماژول را کاهش می‌دهد. اگر هیچ مرجعی به ماژول وجود نداشته باشد، ماژول می‌تواند از حافظه کرنل حذف شود.

در ادامه تابع خواندن پیاده می شود:

```
static ssize_t lifo_read(struct file *file, char __user *buf, size_t
count, loff_t *offset)
{
	struct lifo_char *node;
	ssize_t bytes_read = 0;
	char *temp_buffer;
	int i, ret;

	// Handle non-blocking mode
	if ((file->f_flags & O_NONBLOCK) && list_empty(&lifo_stack))
	return 0;  // Return EOF for empty LIFO

	// Wait until LIFO has data
	if (wait_event_interruptible(lifo_read_queue,
!list_empty(&lifo_stack)))
	return -ERESTARTSYS;

	mutex_lock(&lifo_mutex);

	// Double check LIFO after acquiring lock
	if (list_empty(&lifo_stack)) {
	mutex_unlock(&lifo_mutex);
	return 0;  // Return EOF for empty LIFO
	}

	// Calculate how many bytes to read (min of requested count and
available chars)
	bytes_read = min(count, total_size);
	if (bytes_read == 0) {
	mutex_unlock(&lifo_mutex);
```

```
        return 0;
        }

        // Allocate temporary buffer for reading
        temp_buffer = kmalloc(bytes_read, GFP_KERNEL);
        if (!temp_buffer) {
        mutex_unlock(&lifo_mutex);
        return -ENOMEM;
        }

        // Read characters from LIFO into temporary buffer
        for (i = 0; i < bytes_read && !list_empty(&lifo_stack); i++) {
        node = list_first_entry(&lifo_stack, struct lifo_char, list);
        temp_buffer[i] = node->data;
        list_del(&node->list);
        kfree(node);
        total_size--;
        }

        // Copy data to user space
        ret = copy_to_user(buf, temp_buffer, bytes_read);
        kfree(temp_buffer);

        if (ret) {
        mutex_unlock(&lifo_mutex);
        return -EFAULT;
        }

        mutex_unlock(&lifo_mutex);
        return bytes_read;
}
```

آرگومان های این تابع موارد زیر را شامل می شود:

- struct file *file: مدیریت اطلاعات فایل بازشده (مثل پرچم‌ها، داده خصوصی).
- char __user *buf: اشاره‌گر به بافر کاربر برای کپی داده‌ها.
- size_t count: تعداد بایت‌های مورد درخواست برای خواندن.
- loff_t *offset: جابه‌جایی در موقعیت فایل (معمولاً نادیده گرفته می‌شود).

در این تابع گام های زیر طی می شود:

1. **مدیریت حالت غیر انسدادی (Non-blocking):**
   - اگر فایل در حالت non-blocking باز شده باشد (O_NONBLOCK) و LIFO خالی باشد (list_empty(&lifo_stack))، تابع بلافاصله مقدار 0 را برمی‌گرداند. این نشان‌دهنده پایان فایل (EOF) است.

2. **انتظار برای داده در LIFO:**
   - اگر LIFO خالی باشد، فرآیند به صف انتظار lifo_read_queue اضافه می‌شود تا زمانی که داده‌ای به LIFO اضافه شود. این انتظار با استفاده از wait_event_interruptible پیاده‌سازی شده است.
3. **قفل کردن Mutex برای ایمنی:**
   - mutex_lock برای جلوگیری از دسترسی همزمان به LIFO استفاده می‌شود. این قفل تضمین می‌کند که عملیات خواندن به صورت ایمن انجام شود.
4. **بررسی دوباره LIFO:**
   - پس از گرفتن قفل، دوباره بررسی می‌شود که LIFO خالی نباشد، زیرا ممکن است شرایط بین زمان بررسی اولیه و گرفتن قفل تغییر کرده باشد.
5. **محاسبه تعداد بایت‌های قابل خواندن:**
   - تعداد بایت‌هایی که می‌توان خواند، حداقل بین مقدار درخواست‌شده توسط کاربر (count) و تعداد بایت‌های موجود در LIFO (total_size) است.
6. **اختصاص بافر موقت:**
   - با استفاده از kmalloc، یک بافر موقت در فضای کرنل تخصیص داده می‌شود تا داده‌ها از LIFO به آن کپی شوند.
7. **خواندن داده‌ها از LIFO:**
   - داده‌ها به صورت کاراکتر به کاراکتر از LIFO برداشته می‌شوند. هر گره از لیست حذف می‌شود و حافظه آن آزاد می‌گردد. در این فرآیند، مقدار total_size کاهش داده می‌شود.
8. **کپی داده‌ها به فضای کاربر:**
   - داده‌ها از بافر موقت به بافر کاربر (buf) کپی می‌شوند. اگر خطایی در این کپی رخ دهد (copy_to_user)، مقدار -EFAULT بازگردانده می‌شود.
9. **آزادسازی منابع:**
   - پس از اتمام عملیات، بافر موقت آزاد می‌شود و Mutex نیز آزاد می‌شود.
10. **بازگرداندن تعداد بایت‌های خوانده‌شده:**
    - در نهایت، تعداد بایت‌هایی که از LIFO خوانده‌شده‌اند، به کاربر بازگردانده می‌شود.

تابع نوشتن به این صورت است:

```
static ssize_t lifo_write(struct file *file, const char __user *buf,
size_t count, loff_t *offset)
{
	struct lifo_char *node;
	ssize_t bytes_written = 0;
	int i, ret;
	char *temp_buffer;

	// Check if the requested write size exceeds the maximum LIFO size
	if (count > LIFO_MAX_SIZE)
```

```
	return -EINVAL;  // Invalid argument

	mutex_lock(&lifo_mutex);

	// Check if there's enough space in the LIFO for the requested
write
	if ((total_size + count) > LIFO_MAX_SIZE) {
	mutex_unlock(&lifo_mutex);
	return -ENOSPC;  // No space left on device
	}

	// Allocate temporary buffer for copying data from user space
	temp_buffer = kmalloc(count, GFP_KERNEL);
	if (!temp_buffer) {
	mutex_unlock(&lifo_mutex);
	return -ENOMEM;  // Out of memory
	}

	// Copy data from user space to temporary buffer
	ret = copy_from_user(temp_buffer, buf, count);
	if (ret) {
	kfree(temp_buffer);
	mutex_unlock(&lifo_mutex);
	return -EFAULT;  // Bad address
	}

	// Write data from temporary buffer into the LIFO
	for (i = 0; i < count; i++) {
	node = kmalloc(sizeof(*node), GFP_KERNEL);
	if (!node) {
		kfree(temp_buffer);
		mutex_unlock(&lifo_mutex);
		return -ENOMEM;  // Out of memory
	}
	node->data = temp_buffer[i];
	list_add(&node->list, &lifo_stack);
	bytes_written++;
	total_size++;
	}

	kfree(temp_buffer);
	mutex_unlock(&lifo_mutex);

	// Wake up readers waiting for data
	wake_up_interruptible(&lifo_read_queue);
```

```
	return bytes_written;
}
```

آرگومان ها مانند بخش خواندن است و در تابع گام های زیر طی می کند:

1. **بررسی اندازه داده ورودی:**
   - اگر تعداد بایت‌هایی که کاربر می‌خواهد بنویسد (count) بیشتر از حداکثر اندازه مجاز LIFO (LIFO_MAX_SIZE)) باشد، تابع با مقدار -EINVAL (آرگومان نامعتبر) خاتمه می‌یابد.
2. **قفل کردن Mutex برای ایمنی:**
   - قفل lifo_mutex گرفته می‌شود تا از دسترسی همزمان به LIFO جلوگیری شود.
3. **بررسی فضای خالی در LIFO:**
   - اگر فضای باقیمانده در LIFO برای داده‌های جدید کافی نباشد، تابع با مقدار -ENOSPC (فضای کافی نیست) خاتمه می‌یابد.
4. **اختصاص بافر موقت:**
   - یک بافر موقت در فضای کرنل (temp_buffer) با اندازه count تخصیص داده می‌شود تا داده‌ها از فضای کاربر به آن کپی شوند.
5. **کپی داده‌ها از فضای کاربر:**
   - داده‌ها از فضای کاربر (buf) به بافر موقت کپی می‌شوند. اگر این عملیات با خطا مواجه شود، تابع با مقدار -EFAULT خاتمه می‌یابد.
6. **نوشتن داده‌ها به LIFO:**
   - داده‌ها از بافر موقت به صورت کاراکتر به کاراکتر به LIFO اضافه می‌شوند. هر کاراکتر در یک گره جدید (struct lifo_char) ذخیره می‌شود که در لیست LIFO قرار می‌گیرد.
7. **آزادسازی منابع و Mutex:**
   - بافر موقت آزاد می‌شود و Mutex نیز باز می‌شود.
8. **بیدار کردن خوانندگان منتظر:**
   - اگر فرآیندی در صف انتظار lifo_read_queue منتظر داده باشد، با استفاده از wake_up_interruptible بیدار می‌شود.
9. **بازگرداندن تعداد بایت‌های نوشته‌شده:**
   - تعداد بایت‌هایی که به LIFO نوشته شده‌اند به کاربر بازگردانده می‌شود.

تابع خروج رو به این صورت پیاده می کنیم:

```
static void __exit lifo_exit(void)
{
	struct lifo_char *node, *tmp;

	// Free all nodes in the LIFO
	list_for_each_entry_safe(node, tmp, &lifo_stack, list) {
	list_del(&node->list);
```

```
        kfree(node);
        }

        // Clean up devices and resources
        device_destroy(lifo_class, MKDEV(MAJOR(dev_number), 1));
        device_destroy(lifo_class, dev_number);
        cdev_del(&lifo_write_cdev);
        cdev_del(&lifo_read_cdev);
        class_destroy(lifo_class);
        unregister_chrdev_region(dev_number, 2);

        printk(KERN_INFO "New LIFO Driver: unloaded successfully\n");
}
```

برای خروج گام های زیر طی می شود:

1. **آزادسازی گره‌های LIFO:**
   - لیست lifo_stack بررسی می‌شود و تمام گره‌های موجود در آن (که توسط کاربران قبلاً اضافه شده‌اند) حذف می‌شوند و حافظه آن‌ها نیز آزاد می‌شود.
   - این کار با استفاده از ماکروی list_for_each_entry_safe انجام می‌شود تا از تغییرات در لیست در حین تکرار جلوگیری شود.
2. **حذف دستگاه‌ها و منابع:**
   - گره‌های مربوط به دستگاه‌های read و write با استفاده از device_destroy حذف می‌شوند.
   - دستگاه‌های کاراکتری (cdev) برای خواندن و نوشتن با استفاده از cdev_del آزاد می‌شوند.
   - کلاس دستگاه که با class_create ساخته شده بود، با class_destroy حذف می‌شود.
   - شماره‌های دستگاه که با alloc_chrdev_region تخصیص داده شده بودند، با unregister_chrdev_region آزاد می‌شوند.

```
static int __init lifo_init(void)
{
        int ret;

        printk(KERN_INFO "New LIFO Driver: initializing\n");

        // Initialize LIFO list
        INIT_LIST_HEAD(&lifo_stack);

        // Allocate device numbers with 0 for dynamic allocation
        ret = alloc_chrdev_region(&dev_number, 0, 2, "lifo");
        if (ret < 0) {
```

```
      printk(KERN_ERR "New LIFO Driver: failed to allocate device
numbers\n");
      return ret;
      }

      // Create device class
      lifo_class = class_create(CLASS_NAME);
      if (IS_ERR(lifo_class)) {
      printk(KERN_ERR "New LIFO Driver: failed to create device
class\n");
      unregister_chrdev_region(dev_number, 2);
      return PTR_ERR(lifo_class);
      }

      // Initialize and register read device
      cdev_init(&lifo_read_cdev, &lifo_read_fops);
      ret = cdev_add(&lifo_read_cdev, dev_number, 1);
      if (ret < 0) {
      printk(KERN_ERR "New LIFO Driver: failed to add read device\n");
      class_destroy(lifo_class);
      unregister_chrdev_region(dev_number, 2);
      return ret;
      }

      // Create read device node
      lifo_read_device = device_create(lifo_class, NULL, dev_number,
NULL, DEVICE_NAME_READ);
      if (IS_ERR(lifo_read_device)) {
      printk(KERN_ERR "New LIFO Driver: failed to create read device
node\n");
      cdev_del(&lifo_read_cdev);
      class_destroy(lifo_class);
      unregister_chrdev_region(dev_number, 2);
      return PTR_ERR(lifo_read_device);
      }

      // Initialize and register write device
      cdev_init(&lifo_write_cdev, &lifo_write_fops);
      ret = cdev_add(&lifo_write_cdev, MKDEV(MAJOR(dev_number), 1), 1);
      if (ret < 0) {
      printk(KERN_ERR "New LIFO Driver: failed to add write device\n");
      device_destroy(lifo_class, dev_number);
      cdev_del(&lifo_read_cdev);
      class_destroy(lifo_class);
      unregister_chrdev_region(dev_number, 2);
      return ret;
```

```
    }

    // Create write device node
    lifo_write_device = device_create(lifo_class, NULL,
MKDEV(MAJOR(dev_number), 1), NULL, DEVICE_NAME_WRITE);
    if (IS_ERR(lifo_write_device)) {
    printk(KERN_ERR "New LIFO Driver: failed to create write device
node\n");
    cdev_del(&lifo_write_cdev);
    device_destroy(lifo_class, dev_number);
    cdev_del(&lifo_read_cdev);
    class_destroy(lifo_class);
    unregister_chrdev_region(dev_number, 2);
    return PTR_ERR(lifo_write_device);
    }

    printk(KERN_INFO "New LIFO Driver: initialized successfully with
major number %d\n", MAJOR(dev_number));
    printk(KERN_INFO "New LIFO Driver: created devices /dev/%s and
/dev/%s\n", DEVICE_NAME_READ, DEVICE_NAME_WRITE);

    return 0;
}
```

تابع init ماژول به شکل بالا است گام های آن در ادامه آماده در هر گام بررسی می شود که اگر عملیات موفق نبود موارد قبلی را حذف کند.

1. **شروع فرآیند و چاپ پیام:**
   - پیام اولیه‌ای در لاگ کرنل چاپ می‌شود تا نشان دهد فرآیند بارگذاری ماژول آغاز شده است.
2. **تنظیم لیست LIFO:**
   - لیست LIFO با استفاده از ماکروی INIT_LIST_HEAD مقداردهی اولیه می‌شود.
3. **تخصیص شماره‌های دستگاه:**
   - شماره‌های دستگاه به صورت داینامیک تخصیص داده می‌شوند. اگر این عملیات با شکست مواجه شود، مقدار خطا بازگردانده می‌شود.
4. **ایجاد کلاس دستگاه:**
   - یک کلاس دستگاه با استفاده از class_create ایجاد می‌شود. اگر این عملیات شکست بخورد، شماره‌های تخصیص داده شده آزاد می‌شوند.
5. **ثبت دستگاه خواندن:**
   - ساختار cdev برای دستگاه خواندن مقداردهی اولیه می‌شود و با استفاده از cdev_add ثبت می‌شود. در صورت بروز خطا، منابع تخصیص داده شده آزاد می‌شوند.
   - یک گره دستگاه برای خواندن (/dev/lifo_read) ایجاد می‌شود.

6. **ثبت دستگاه نوشتن:**
   - مشابه دستگاه خواندن، ساختار cdev برای دستگاه نوشتن مقداردهی اولیه و ثبت می‌شود.
   - یک گره دستگاه برای نوشتن (/dev/lifo_write) ایجاد می‌شود.

بررسی عملکرد

برای بررسی عملکرد یک برنامه ساده c نوشتیم که با ماژول تعامل می کند. ۳ تا حالت داریم:

1. یکبار بنویسیم و یک بار بخوانیم.
2. چند بار بنویسیم و یک بار بخوانیم.
3. حالت دستی

نتایج ۳ تست در ادامه آمده:

```
➜  P5 git:(main) make
make -C /lib/modules/6.11.0-25-generic/build
M=/home/mvajhi/code/essential-computing-skills/CA2/P5 modules
make[1]: Entering directory '/usr/src/linux-headers-6.11.0-25-generic'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-13 (Ubuntu
13.3.0-6ubuntu2~24.04) 13.3.0
  You are using:        gcc-13 (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
  CC [M]
/home/mvajhi/code/essential-computing-skills/CA2/P5/lifo_driver.o
  MODPOST
/home/mvajhi/code/essential-computing-skills/CA2/P5/Module.symvers
  CC [M]
/home/mvajhi/code/essential-computing-skills/CA2/P5/lifo_driver.mod.o
  LD [M]
/home/mvajhi/code/essential-computing-skills/CA2/P5/lifo_driver.ko
  BTF [M]
/home/mvajhi/code/essential-computing-skills/CA2/P5/lifo_driver.ko
Skipping BTF generation for
/home/mvajhi/code/essential-computing-skills/CA2/P5/lifo_driver.ko due
to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-6.11.0-25-generic'
gcc test_lifo.c -o test_lifo.out
➜  P5 git:(main) insmod lifo_driver.ko
➜  P5 git:(main) ./test_lifo.out 1
Test mode 1: Write then read
Opening /dev/lifo_write
Writing: Hello LIFO driver!
Wrote 18 bytes
Opening /dev/lifo_read
Read 18 bytes: !revird OFIL olleH
Reading from empty buffer...
```

```
Correctly received EOF (0 bytes) from empty buffer
➜  P5 git:(main) ./test_lifo.out 2
Test mode 2: LIFO behavior demonstration
Opening /dev/lifo_write
Writing: hi
Wrote 2 bytes
Writing: bye
Wrote 3 bytes
Opening /dev/lifo_read
Reading from device (should be in LIFO order):
Read 5 bytes: eybih
SUCCESS! Characters were correctly stored in LIFO order.
➜  P5 git:(main) ./test_lifo.out
LIFO Character Device Tester
============================
Select an option:
1. Write data to LIFO
2. Read data from LIFO
3. Exit

Enter option (1-3): 1
Enter data to write:  1 2
Wrote 5 bytes

Enter option (1-3): 1
Enter data to write:  3 4
Wrote 5 bytes

Enter option (1-3): 2
Read 10 bytes:  4 3  2 1

Enter option (1-3): 3
Exiting...
```

سوالات تشریحی

انواع درایور ها

- **Block Device Driver:**
  - این درایورها برای دستگاه‌هایی استفاده می‌شوند که داده‌ها را به صورت بلوک‌های بزرگ پردازش می‌کنند.
  - مثال‌ها: هارد دیسک‌ها، فلش مموری‌ها.
  - در این نوع، دسترسی تصادفی (random access) به داده‌ها امکان‌پذیر است.

- **Network Device Driver:**
  - این درایورها برای دستگاه‌هایی که از شبکه استفاده می‌کنند، مانند کارت‌های شبکه (NIC) طراحی شده‌اند.
  - این درایورها معمولاً داده‌ها را به صورت بسته‌های شبکه‌ای (packets) مدیریت می‌کنند.
- **Virtual Device Driver:**
  - این نوع درایورها برای دستگاه‌های مجازی که به طور مستقیم به سخت‌افزار متصل نیستند استفاده می‌شوند.
  - مثال‌ها: درایورهای سیستم فایل مجازی (مانند tmpfs).

Major, Minor

Major Number:

- این شماره نشان‌دهنده نوع دستگاه است و کرنل از آن برای شناسایی درایور مرتبط با دستگاه استفاده می‌کند.
- به عنوان مثال، تمام دستگاه‌هایی که به یک نوع درایور خاص مربوط می‌شوند، یک Major Number یکسان دارند.
- مثال: Major Number شماره 8 ممکن است به دیسک‌های SCSI تعلق داشته باشد.

Minor Number:

- این شماره نشان‌دهنده یک دستگاه خاص در دسته‌ای از دستگاه‌ها است که توسط یک درایور مدیریت می‌شوند.
- به عنوان مثال، اگر چندین هارد دیسک SCSI داشته باشید، هرکدام یک Minor Number منحصربه‌فرد دارند.

مشکلاتی که مواجه شدم

https://chatgpt.com/share/68205881-1348-8001-9bcd-d9721847b045

یک مشکلی که با آن مواجه شدیم وقتی از لایسنس bsd استفاده می کردم ارور می داد که باید از gpl استفاده کنیم. نمی دونم ولی احتمالا این مشکل به این دلیل بود که از توابع و کتابخانه هایی با لایسنس gpl در کد استفاده کردم.

مشکلی دیگری که با آن مواجه شدم این بود که ماژول با موفقیت کامپایل می شد ولی نمی توانستم به کرنل اضافه کنم. دلیل این موضوع احتمالا به دلیل ناهمخوانی کامپایلر gcc با نسخه کرنل بود و با نصب مجدد نسخه مناسب مشکل بر طرف شد.