پردیس علوم
دانشکده ریاضی، آمار و علوم کامپیوتر

# پوسته یونیکس

نگارنده

**نجمه دستورانی**

استاد راهنما : مهندس ابراهیم نقیب زاده مشایخ

پایان نامه برای دریافت درجه کارشناسی
در رشته علوم کامپیوتر

تابستان ۱۳۹۷

# چکیده

پوسته یونیکس یک مفسر خط فرمان یا شل است که یک واسط کاربرسنتی را برای سیستم عامل های یونیکس وشبه یونیکس مهیا می کند.  در این پروژه قصد داریم به بیان مفهوم دقیق پوسته یونیکس،انواع پوسته وهمچنین اهمیت پوسته بپردازیم.

# پیشگفتار

به احتمال زیاد اکثر افراد تاکنون فقط با محیط های بارابط گرافیکی(GUI) کار کرده اند اما رابط کاربری دیگری که لینوکس وسایر سیستم عامل های شبه یونیکس یا بر پایه یونیکس آن را دارند رابط خط فرمانCommande Line Interface است که به اختصار CLI می نامند.قدرت حقیقی لینوکس ویونیکس در این رابط نهفته است.که به مدت ۳۰ سال است که یونیکس را به عنوان قدرتمندترین سیستم عامل مطرح کرد واکنون این قدرت نهفته درتمام سیستم های شبیه یا بر پایه یونیکس مانند لینوکس وجود دارد. زمانی می توان با یک سیستم عامل راحت تر کار کرد که امکان کارباخط فرمان درآن فراهم شده باشد و در لینوکس این مهم صورت گرفته است. به دلیل اهمیت پوسته یونیکس ما در این پروژه بامطالعه منابع مختلف دراین باب سعی در معرفی هرچه دقیق تر پوسته یونیکس وویژگی های آن وچگونگی کار با آن را داریم. در ابتدا درباره سیستم عامل یونیکس وتاریخچه ونحوه به وجود آمدن آن مطالبی را بیان خواهیم کرد.وسپس با مفهوم دقیق پوسته ونحوه عملکرد آن آشنا خواهیم شد.وسپس به تفاوت بین پوسته وهسته خواهیم پرداخت.پس از آن به سیر تکامل پوسته های یونیکس ولینوکس خواهیم پرداخت.وپس ازآن انواع پوسته با تاریخچه آن ها را بیان خواهیم نمودو... در آخر کد برخی پوسته ها را خواهیم داشت.

# فهرست مطالب

# فصل ۱

# سیستم عامل یونیکس

یونیکس یک سیستم عامل با قابلیت چندکارگی وچند کاربره است که در سال۱۹۶۹به دست گروهی از کارمندان آزمایشگاه های بل متعلق به شرکت تلفن وتلگراف آمریکا شروع به توسعه شد.یونیکس درآغاز به زبان اسمبلی نوشته شد اما در سال ۱۹۳۷ به طور کلی به زبان سی بازنویسی شد،که این کار توسعه یونیکس وپورت کردن آن به دیگر platform (سکو)ها را ساده ترمی کرد.

## ۱.۱ پیشینه

درسال ۱۹۶۰ نخستین سیستم عامل چند کاربره که به عنوان خدمتگزار می توانست با نصب روی یک رایانه به چند رایانه دیگر سرویس دهد(اشتراک زمانی)، ساخته شد. این سیستم عامل که سی تی اس اس نام داشت و می توانست ۲۰ کاربر را با یک رایانه ۷۰۹۰ آی بی ام به خوبی اداره کند به شدت مورد توجه ومفید واقع شد. به طوریکه در سال ۱۹۶۵ سه شرکت از برجسته ترین نقش آفرینان رایانه در جهان بر آن شدند که روی سیستم عامل مولتیکس کار کنند ولی چون این سیستم اجزای زیادی داشت واز حد معین بزرگتر شد، یکی از این شرکت ها پروژه را رها کرد.پس از این ماجرا دنیس ریچی ،کن تامسون و برین کرنیگان یونیکس (Unics) را در آزمایشگاه بل براساس مولتیکس به وجود آوردند وبعد ها به تدریج Unix نام گرفت . [۱]

## ۲.۱ بازنوشتن به زبان سی

همانطور که بیان شد یونیکس درابتدا به زبان اسمبلی نوشته شد.امادر ۱۹۷۲،یونیکس به زبان سی بازنوشته شد.این برخلاف پندار همگانی آن زمان بود که می گفت « هرچیز پیچیده ای مثل سیستم عامل که باید با رویدادهای حساس به زمان سرو کار داشته باشد،باید منحصرا به زبان اسمبلی نوشته

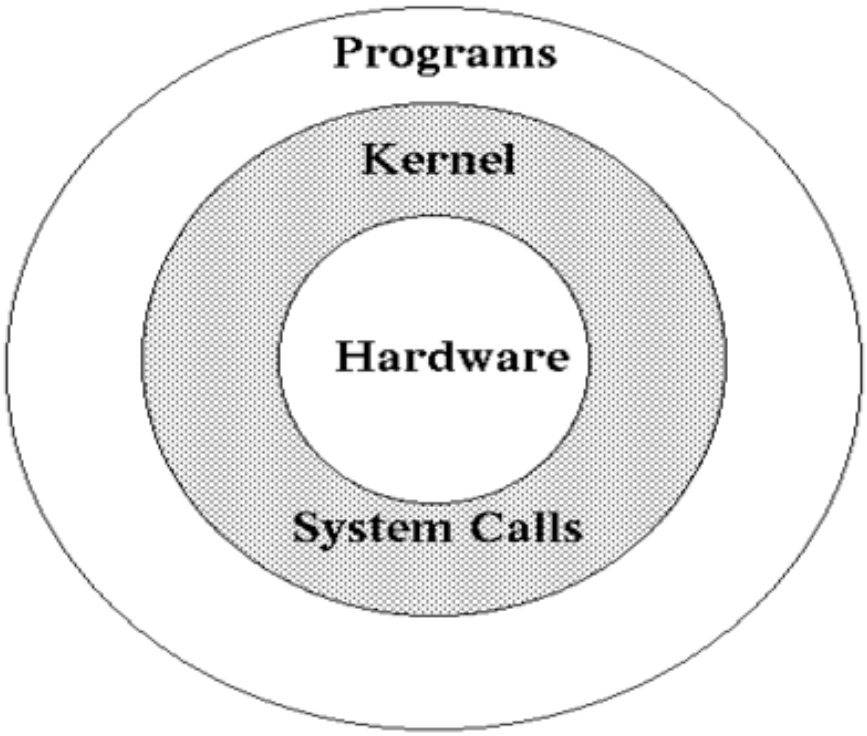
شود.» [۱] نتیجه مهاجرت از زبان اسمبلی به زبان سطح بالاتر سی این بود که کد یونیکس قابل حمل تر شدوبرای اجرا بر روی ماشین های دیگر تنها کافی بود قسمت های کوچکی از آن به زبان اسمبلی مخصوص همان ماشین نوشته شوند.

## ۳.۱ عملکرد سیستم عامل یونیکس در اجرای برنامه

يك سيستم عامل اعمال بسياري را انجام مي‌دهد: از جمله عملاً برنامه را اجرا مي‌كند و تمام ورودي و خروجي را در كامپيوتر كنترل مي‌نمايد. يونيكس اين اعمال را از طريق سه بخش مجزا اما بسيار يكپارچه انجام مي‌دهد: سيستم پرونده ، پوسته ، هسته . [۳] هسته، مسئول تمام اعمال اوليه سيستم عامل مي‌باشد، هسته در اوليه ترين سطح خود، وظيفه اداره حافظه كامپيوتر و چگونگي تخصيص دستورالعملهاي نرم افزاري به اين حافظه، اجراي تمام فرامين، سرپرستي سيستم پرونده، رسيدگي به خطاها، و غيره را به عهده دارد هنگامي كه كامپيوتر روشن مي‌شود، هسته شروع به كار مي‌كند و بدون توجه به نرم‌افزار يا پوسته‌اي كه اجرا مي‌كنيد، در حافظه باقي مي‌ماند. به عنوان يك كاربر، نيازي نيست كه درباره هسته فكر كنيد، تنها بايد بدانيد كه هسته كار مي‌كند. سيستم پرونده به پيگيري پرونده‌ها و محل قرارگيري آنها مي‌پردازد. هر چيزي در يونيكس ، خواه پرونده‌اي كه در يك پرداز شگرمتن ايجاد مي‌شود يا راه‌اندازي كه براي فرستادن دستورالعملها به چاپگر استفاده مي‌گردد، در پرونده‌اي شامل مي‌شود. پوسته يا مفسر خط فرمان، بخشي از يونيكس است كه عملاً در اغلب موارد در حال استفاده از آن خواهيد بود. در اصل، پوسته‌ها دستورالعملها را مي‌گيرند و آنها را به دستوراتي قابل فهم توسط هسته تبديل مي‌كنند. هنگامي كه برنامه‌اي را اجرا مي‌كنيد، به پوسته مي‌گوييد كه برنامه‌اي را از هسته اجرا نمايد.

## ۴.۱ مزایای کلی یونیکس

۱. بازه گسترده اي ازرابط هاي كاربر.
(ترمینال های ایکس ویندوز و اسکی) [۳]
۲. برنامه هاي كاربردي توزيع شده وچندكاربره. [۳]
۳. معماري ۶۴ بيت. (افزايش آدرس حافظه از ۴ گيگا بايت تا ۱۶۰۰۰۰۰۰گيگا بايت) [۳]
۴. برابر بااستانداردهاي باز. [۳]
۵. مقياس پذير، قابل حمل و انعطاف پذير. (ازمين فريم ها تا لپ تاپ ها،ازتعدادنودهاي كم تازياد.) [۳]
۶. محيط برنامه نويسي توكار(سی ،ای دبليوكا ،اسكريپت ،پرل ، شل) [۳]




شکل ۱.۱: ساختار کامپیوتر

# فصل ۲

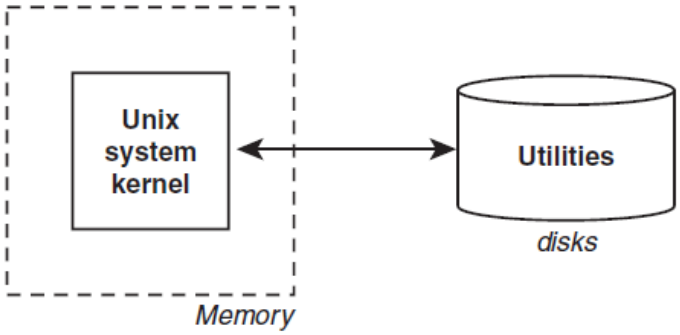
# پوسته

تقریباً هر اختراع قابل استفاده بشر يك نقطه رابط با آنچه انسان متقابلاً عمل مي‌كند دارد. نقطه رابط انسان با يونيكس ، پوسته است كه يك لايه برنامه است و يك محيط براي وارد كردن فرمانها و پارامترها براي توليد يك نتيجه داده شده، براي كاربر فراهم مي‌كند. یک پوسته یونیکس درواقع یک زبان اسکریپت نویسی ویک واسط خط فرمان تعاملی است که همچنین توسط خود سیستم به عنوان تسهیلات برای هدایت(شل اسکریپت)اجرای سیستم استفاده می شود. البته در اصل پوسته یک رابط است میان انسان وهسته ی سیستم عامل که زبان انسان هارا به زبان ماشین تبدیل می کند تا هسته سیستم عامل آن را بفهمد. [۲] (ببینید شکل ۱.۲ ) می توان در یک تجسم غیر رسمی گفت که میز کارهای ویندوز و مکینتاش نیز یک پوسته ی گرافیکی هستند که رابط بین انسان وهسته ی این سیستم عامل هاست و زبان انسان را به زبان ماشین که برای هسته قابل فهم باشد ترجمه می کند. [۲]

در طول مدت به وجود آمدن یونیکس تا انتشار نهایی لینوکس پوسته های بسیاری منتشر شده وپوسته یک سیر تکاملی را پشت سر گذاشته است.که در قسمت بعد راجع به این مطلب بحث خواهد شد.

## ۱.۲ تفاوت هسته با پوسته

هسته در واقع مرکز مدیریت دریک سیستم عامل است که تمامی فرامین برای اجرا شدن از طریق این مرکز مدیریت صادر می شوند. تمامی اجزای سیستم عامل وابسته به این هسته است.هسته به صورت لایه بندی شده کار می کند و هر چقدر به درون این لایه بیشتر وارد شویم به منابع ودسترسی های بیشتری دسترسی پیدا خواهیم کرد.اما نکته اینجاست که این لایه ها یاپوسته ها را ما به عنوان پوسته می شناسیم. پوسته ها در واقع رابط کاربری ما برای دستور دادن به هسته سیستم عامل هستند و ما از طریق این رابط ها می توانیم با سیستم عامل ارتباط برقرار کنیم.در واقع می توان از پوسته




شکل ۱.۲ : سیستم یونیکس

به عنوان یک مترجم نام بردکه دستورات کاربر را دریافت می کند وآنها را به زبان هسته ترجمه می کند تا هسته آنها را اجرا کند. پوسته یک نوع برنامه است که در حالت دسترسی محدود اجرا می شود ودستوراتی که از جانب کاربر دریافت می کند رابه حالت دسترسی بالا منتقل می کند. که هسته اینجا قرار داردو دستورات کاربر در این قسمت اجرا می شوند.
اما چه اتفاقی می افتد اگر یک سیستم عامل داشته باشیم که دارای هسته باشد اما پوسته نداشته باشد؟ پاسخ ساده است، درچنین حالت یک سیستم عامل داریم که فقط وفقط می تواند کارهایی که ازپیش برایش تعریف شده را انجام دهد و به هیچ عنوان نمی تواند درخواست های کاربران را پاسخ دهد زیرا رابط کاربری یا پوسته ای وجود ندارد که بتواند با هسته ارتباط برقرار کند.این دقیقا مثل این است که بگوییم شما ویندوز را نصب کرده اید اما هیچ محیطی برای ارتباط با هسته از لحاظ گرافیکی و خط فرمانی ندارید. اکنون آیا ممکن است فقط پوسته داشته باشیم اما هسته نداشته باشیم؟ پاسخ این سوال مشخص است.چنین چیزی امکان ندارد ،در واقع پوسته محیطی است که خود هسته برای برقراری ارتباط با ما در اختیارمان قرار می دهد ونبود هسته به معنای نبود پوسته است.

## ۲.۲ شل اسکریپت چیست ؟

در حالت عادی شل یک محیط محاوره ای دارد. به این معنا که کاربر دستورات را از طریق صفحه کلید به پوسته می دهد وپوسته دستورات را اجرا می کند.اما اگر به جای اینکه کابر تک تک دستورات را به پوسته جهت اجرا بدهد ، همه دستورات را در یک فایل متنی ذخیره کرده وسپس فایل را جهت اجرا در اختیار پوسته قرار دهد در این صورت کاربر یک شل اسکریپت اجرا کرده است . شل اسکریپت در واقع مجموعه ای از دستورات هستند که در یک فایل متنی ذخیره شده اند.
شل اسکریپت شبیه به یک دسته فایل در MS-DOS می باشد ولی به مراتب قوی تر از آن است.

اکنون این سوال پیش می آید که چرا از شل اسکریپت استفاده می کنیم ؟ شل اسکریپت ها ورودی را از کاربر یا فایل دریافت کرده وخروجی را در صفحه نمایش نشان می دهند.هر کاری که کاربر به صورت تکراری مجبور به انجام آن است را می توان به صورت اسکریپت انجام داد.مانند اجرای اتوماتیک وظایف تکراری. ازجمله مزایای اسکریپت نویسی می توان به چند مورد زیر اشاره کرد :
ـ صرفه جویی در زمان
ـ ساختن ابزارها وبرنامه های دلخواه
ـ سفارشی کردن موارد مربوط به مدیریت سیستم
وهم چنین چون اسکریپت ها به خوبی تست شده اند ، احتمال بروز خطا در هنگام پیکر بندی سرویس ها و وظایف راهبری سیستم مانند اضافه کردن کاربر جدید، به شدت کاهش می یابد.

## ۳.۲ سیر تکامل پوسته های یونیکس و لینوکس

شل رابط کاربر با سیستم عامل خواهد بودوراهی که کاربر از طریق آن می تواند با سیستم ارتباط برقرار کند.پس از اینکه به یک سیستم یونیکسی لاگین کنید در مکانی به نام شل قرار خواهید داشت.معمولا یک صفحه مشکی بانوشته های سفید یک شل است وکاربر درآن دستورات را به صورت متنی یا اسکریپت نویسی پوسته یونیکس وارد می کند که در اسکریپت دستورها یکی پس از دیگری اجرا می شوند. [۶] ازابتدای به وجود آمدن یونیکس انواع مختلفی از شل ها به وجود آمدن که از این قرارند:
۱ ـ تامسون شل: با توجه به تاریخچه ای که در دنیای اینترنتی وجود دارد اولین شل ساخته شده برای یونیکس، تامسون شل بوده است که به وسیله کن تامسون ودر آزمایشگاه های بسیار معروف بل نوشته شده است ودر توزیع های نسخه ۱ تا۶یونیکس از۱۹۷۱تا ۱۹۷۵به کار برده شده است و قابلیت هایی مانند پایپ کردن و کنترل ساختار ها با استفاده از If و Goto را داراست . [۶] با این حال امروزه دیگر این شل با سیستم های یونیکسی ارائه نمی شود وبه نوعی از رده خارج محسوب می شود.
۲ ـ شلPWB:
پوسته پی دبلیو بی یا Mashay Shell یک نسخه به روز شده و البته ویرایش شده از تامسون شل است که توسط جان ماشی نوشته شده است . این شل برای بهتر شدن برنامه نویسی پوسته طراحی شده است وقابلیت های جالبی مانند $Then, Else, EndIf, If$درآن قرار دارد وسوئیچ وساختارWhile نیز برای اولین بار در این پوسته معرفی شدند. [۶]
۳ـ پوسته بورن یاsh
پیشرفت های یونیکس با روی کار آمدن "بورن شل" کم کم بیشتر شد . بورن شل در نسخه ۷ یونیکس که در۱۹۷۹ منتشر شد استفاده شد. این شل امکانات پایه ای داشت وپس از آن در بیشتر شل های یونیکس از آن استفاده شد . در اصل این پوسته در آزمایشگاه بل توسط استیون بورن برای سیستم عامل $AT \& T\ UNIX$نوشته شد.البته در لینوکس چندان از این پوسته استفاده نمی شود ودر سایر نسخه های یونیکس مانند Free BSD نیز این پوسته بایک نسخه به نام POSIX

جایگزین شده است. POSIX از ویرایشگر خط فرمان،تاریخچه فرمان(که نه در پوسته بورن بود ونه در سایر مفسر های خط فرمان مانندDos ) وهمینطور اسامی مستعار به طور پیش فرض پشتیبانی می کرد. [۲] برنامه های بورن شل با نام sh شناخته می شوند ودر مسیر /bin/sh قرار داده می شوند. در بیشتر سیستم ها بورن شل ممکن است به شل های معرفی شده در زیر لینک شده باشد:

Ash -

Bash -

Zsh -

Ksh -

یک مثال ساده:

$$\#!/bin/sh$$

$$1 - "!echo" Hello\ World$$

$$2 - !"echo" Hello\ World$$

۴ـ آلکموییست شل

شل آلکموییست همچنین به نام $AShell$نیز شناخته می شود و یک شل یونیکسی سبک است که توسط آقای آلکموییست ودر حدود سال های ۱۹۸۰نوشته شده است. این شل نوع دیگری از بورن شل است که در نسخه های BSD که در سال های ۱۹۹۰عرضه می شده جایگزین بورن شده است . در برخی نسخه های دبیان و ابونتو نیز این شل با نام dash شناخته می شود که مخفف $debian\ almquist\ shell$ است. همچنین آلکموییست در سیستم های "امبدد " از محبوبیت بسیاری برخوردار است. [۶] این شل سریع و کوچک و سازگار با استاندارد های پوزیکس است و این دلیل خوبی است که چرا این شل در دستگاه های "Embeded" محبوب است. ASH از مکانیسم های تاریخچه دستورات پشتیبانی نمی کند اما ممکن است نسخه های اخیر آن با توجه به پیشرفت هایی که کرده است تاریخچه را نیز ساپورت کنند.

۵ـ C Shell یا CSh

این پوسته که اغلب در BSD ها استفاده می شود توسط گروه توسعه دهندگان نرم افزار دانشگاه برکلی که از محدودیت های پوسته بورن به ستوه آمده بودند ایجاد شد. دلیل نامگذاری این پوسته به C ،گرامر یا Syntax این پوسته است که به زبان برنامه نویسی C بسیار شباهت دارد واین موضوع خود باعث دشواری در نوشتن برنامه های پوسته می شود. [۲] (این برنامه ها درون خود پوسته اجرا می شوندو برای اجرا حتما به پوسته مخصوص به خود احتیاج دارند درواقع پوسته این برنامه هارا تفسیر می کند.)در این پوسته ،سیستم کنترل فعالیت ها وتاریخچه خط فرمان به صورت پیشرفته تر و کامل تر وجود دارد.البته در حال حاضر در Free BSD ها نسخه های پیشرفته Csh با tsh جایگزین شده که tsh در قسمت های بعد مورد بررسی قرار می گیرد.البته در لینوکس می توان از هردوی این پوسته ها استفاده کرد.

۶ـ Korn Shell یا Ksh یا Pdksh

شرکت AT and T ، پوسته Ksh را در سال ۱۹۸۶ منتشر کرد، این پوسته که توسط دیوید کورن نوشته شد ، پاسخی بود برای انتشار CShell . این پوسته مانند CShell از کنترل فعالیت ،تاریخچه

خط فرمان واسامی مستعار پشتیبانی می کرد ونسبت به پوسته بورن ،کاربر پسند تر بود.دراین نسخه برنامه نویسی نیز بسیار راحت تر شده بود و همین طور ابزار های آن نیز بسیار بیشتر شده بود. Pdksh نیز یک پوسته کورن است با این تفاوت که یک نسخه ی حوزه عمومی می باشد که از ابتدایpd آن نیز می توان متوجه این موضوع شد. [۲]

۷ـ bash

بش یک پوسته سازگار ونه برپایه ی sh است که توسط توسعه دهندگان بنیاد نرم افزار (FSF) توسعه داده شد. بش پوسته پیش فرض اغلب لینوکس هاست. این پوسته شبیه به پوسته ksh است با ابزارها وقابلیت های بیشتر ازجمله قابلیت های جدید این پوسته راهنمای درونی آن است.همین طور ویرایش مستقیم خط فرمان ویا ویرایش تاریخچه خط فرمان و جستجو با کلید های جهتدار ویا متغیرهای محیطی بسیار زیاد آن. البته این پوسته درنسخه های تجاری یونیکس موجود نیست ودلیل آن هم مجوز بسیار عالی جی ان یو/جی پی ال است. بش یکی از محبوب ترین وپراستفاده ترین پوسته ها در بین انواع پوسته است. این پوسته برای کاربرانی که دارای مسیر /bin/bash هستند نصب واستفاده شده است.همچنین تاریخ عرضه آن سال ۱۹۸۹ بوده است. [۶]

بش از دستورات بسیار گسترده ای استفاده می کند. اسکریپت های بش نیز با دستور زیر شروع می شود : $\#/bin/bash$

بش می تواند دستورات را از روی فایل خوانده وحتی خروجی را بر روی فایل و پایپ لاین ها نمایش می دهد.در زیر یک مثال ساده از اسکریپت های بش را مشاهده می کنیم:

$\#!bin/sh$

$if$ [ $\$days - gt$ 365]

۱۰

$then$

$echo\ This\ is\ over\ a\ year.$

fi

۸ـ Tcsh Shell

این پوسته یک پوسته ی پیشرفته براساس پوسته ی CShell است. کلمه T در ابتدای نام آن اول نام سیستم عامل TENEX است که بر روی DEC PDP-۱۰ اجرا می شود.در این پوسته به CShell قابلیت های بیشتری اضافه شده است مانند ویرایش تاریخچه به سبک bash و یا پرسش های قبل اعمال تغییرات مانند پرسش قبل از پاک کردن یک شاخه (Folder) . [۲] البته همانطور که قبلا اشاره کردیم Free BSD های پیشرفته از این پوسته به عنوان پوسته ی پیش فرض استفاده می کنند.

۹ـ زی شل یا ZSH

Paul Falstad اولین نسخه از زی شل رادر سال ۱۹۹۰ نوشت.این شل را می توان دریونیکس به عنوان یک شل فعال و پویا استفاده کرد که مترجمی بسیار قوی برای شل اسکریپتینگ دارد.زی شل نسخه توسعه داده شده بش است که بهبود های بسیارزیاد در آن انجام شده است. این نوع شل دارای قابلیت های بسیار خوبی است که در زیر آورده ایم:

ـ تصحیح و تکمیل خودکار دستورات با قابلیت برنامه ریزی
ـ اشتراک گذاری تاریخچه دستورات درتمام شل های در حال اجرا
ـ بهبود مدیریت متغیرها و آرایه ها
ـ وتغییرات بسیاری دیگر که این شل را به شلی بسیار قدرتمند وجالب تبدیل کرده است.
یکی از معروف ترین کانفیگ ها ودلایل اصلی محبوبیت بالای زی شل وجود Oh My Zsh است که این شل را تبدیل به شلی بسیار کار آمد وبی نظیر در نوع خود می کند.

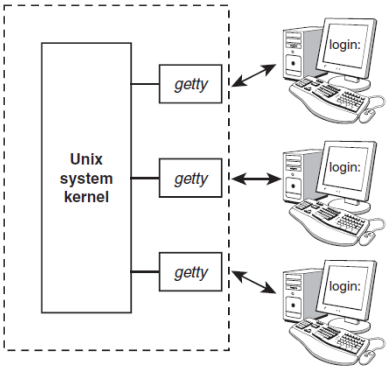
## ۴.۲ کدام پوسته ؟

همانطور که قبلا اشاره کردیم اغلب لینوکس ها از bash به عنوان پوسته پیش فرض استفاده می کنند و البته لینوکس های کم حجم مثلا در حجم یک فلاپی از پوسته های کم حجم تر استفاده می کنند. از آنجایی که کار با bash بسیار آسان است و مانند کار کردن، برنامه نویسی برای آن نیز راحت تر است اغلب لینوکس کارها این پوسته را می پسندند وبنابراین اغلب اسکریپت های موجود در برنامه های نصب مربوط به لینوکس هم بر اساس این پوسته نوشته می شوند.

# فصل ۳

# چگونگی برقراری ارتباط بین هسته وپوسته

وقتی سیستم عامل یونیکس شروع به کار می کند، برنامه یونیکس (هسته) به داخل حافظه اصلی بار می شود. وتا زمانی که کامپیوتر خاموش شود آنجا می ماند. در طول اجرای یک پردازش ،برنامه init به عنوان یک وظیفه پیش زمینه اجرا می شود و تا زمان خاموش شدن کامپیوتر درحال اجرا می ماند. برنامه init فایل /etc/inittab را اسکن می کند [۵] ، این فایل لیست می کند که چه پورت هایی ترمینال دارند ومشخصاتشان چیست. وقتی یک ترمینال باز و فعال پیدا می شود init، برنامه getty را صدا می زند که getty یک login را صادر می کند و روی مانیتور همان ترمینال باز، نمایش می دهد.با این فرآیند ها ی درحال اجرا سیستم برای ارتباط با کاربر آماده است. بنابر آنچه قبلا گفته ایم ارتباط کاربر با سیستم از طریق پوسته برقرار می شود. درواقع کاربر با تایپ دستوراتی در پوسته با هسته سیستم یونیکس ارتباط برقرا می کند. (ببینید شکل۱.۳ )

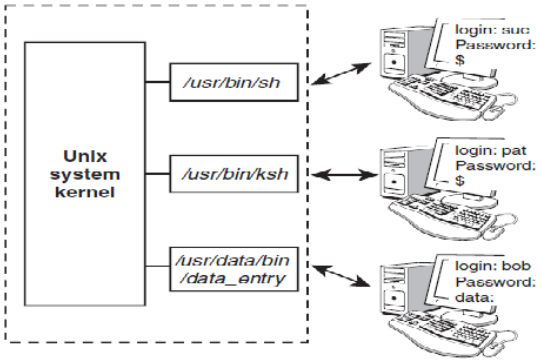



شکل ۱.۳ : فرآیند getty

## ۱.۳ برقراری ارتباط بین هسته و پوسته

زمانی که getty یک login صادرمی کند کاربر در یک پوسته فعال آماده فرمان دادن به سیستم است. در طول login کردن وقتی کاربر " نام کاربری" خود را وارد می کند getty یک رمز صادر می کند . بعد از اینکه کاربر " کلمه رمز " خود را تایپ کرد ، getty ، login را صدا می زند تا " کلمه رمز " وارد شده را چک کند ، login برای این کار فایل $/etc/passwd$را اسکن می کند. [۵] (ببینید شکل ۲.۳ ) اگر کلمه رمز تایپ شده توسط کاربر درست باشد ، login کاربر را به دایرکتوری خانه کاربر می برد. " نام کاربر" و " کلمه رمز " وارد شده توسط $/etc/passwd$ مشخص می شود. در این زمان ، پوسته یک فرمان$ (prompt) را صادر می کند. وقتی پوسته تمام می شود هسته کنترل را به برنامه init برمی گرداند. تمام شدن پوسته از دو طریق امکان پذیر است :

۱ـ با فرمان خروج . ۲ـ فرستادن فرمان kill از طرف هسته برای فرآیند پوسته.




شکل ۲.۳: سه کاربر login شدند.

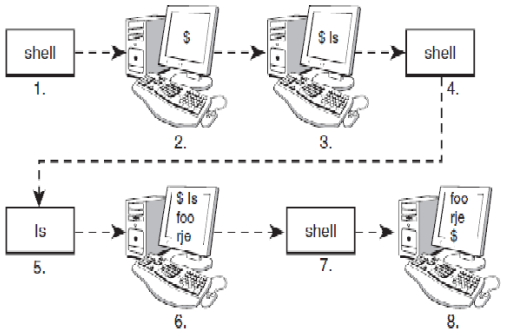
## ۲.۳ توابع و ویژگی های پوسته

تمامی پوسته ها ایجاد شده ، یک هدف مشترک دارند و آن هم ایجاد واسط کاربری برای یونیکس است . برای ایجاد این امکان تمامی پوسته ها مشخصات پایه ای زیر را پیشنهاد می کنند :

- مفسر خط فرمان [۵]
- کلمات رزرو شده [۵]
- $wildcards$ پوسته [۵]
- دستیابی وراه اندازی فرمان های برنامه [۵]
- راه اندازی فایل : دوباره جهت دادن ورودی /خروجی و pipes [۵]
- نگهداری از متغیرها [۵]
- کنترل محیط [۵]
- برنامه نویسی پوسته [۵]

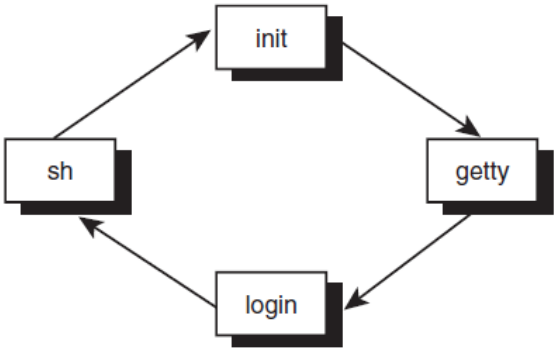
## ۳.۳ تایپ کردن دستورات در پوسته

وقتی پوسته شروع به کار می کند ، ابتدا یک $ در ترمینال کاربر نمایش می دهد وسپس منتظر تایپ دستورات توسط کاربر می ماند.( مرحله ۱ و۲ درشکل ۳.۳ ) وقتی که کاربر دستوری را وارد کرده و کلید Enter را فشار می دهد(مرحله ۳ در شکل ۳.۳ ) ، پوسته شروع به تجزیه و تحلیل دستور تایپ شده می کند.(مرحله ۴ درشکل ۳.۳ ) اگر کاربر فراخوانی یک برنامه خاص را از پوسته درخواست کند، پوسته Disk را جستجو می کند. درواقع پوسته تمام دایرکتوری هایی که کاربر در PATH خود مشخص کرده را برای یافتن برنامه مورد نظر کاربر، جستجو می کند.به محض پیدا شدن برنامه مورد نظر ،پوسته یک زیر پوسته می سازد واز هسته در خواست می کند که این زیر پوسته را با برنامه




شکل ۳.۳: چرخه دستور

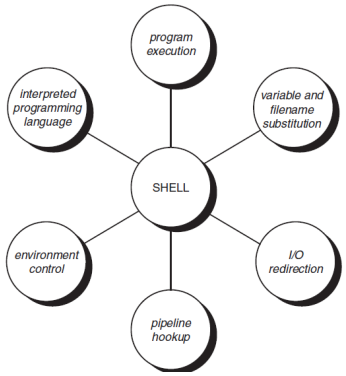
خواسته شده جایگزین کند.سسپس login پوسته sleep می شود تا برنامه تمام شود. [۵] (مرحله ۵ درشکل ۳.۳ ) هسته ،برنامه مشخص شده را در حافظه کپی کرده و سپس شروع به اجرای آن می کند.این برنامه کپی شده را " فرآیند " می نامند.در این روش بین برنامه ذخیره شده در یک فایل در disk و یک فرآیند ذخیره شده در حافظه که خط به خط اجرا می شود تمایز وجود دارد. اگر برنامه خروجی تولید کند ، این خروجی روی ترمینال کاربر نمایش داده می شود مگر اینکه این خروجی به داخل دستور دیگری هدایت شود. مشابها اگر برنامه ، ورودی از روی ورودی استاندارد بخواند ،منتظر می ماند تا شما ورودی را تایپ کنید مگر اینکه ورودی از یک فایل یا از دستور دیگری تامین شود.(مرحله ۶ درشکل ۳.۳ ) وقتی دستور " اتمام " اجرا می شود کنترل دوباره به login پوسته برمی گردد تا پوسته برای دستور بعدی کاربر فعال شود.(مرحله ۷ و ۸ درشکل ۳.۳ ) این چرخه تا زمانی که کاربر در login قرار دارد ادامه پیدا می کند . وقتی کاربر سیستم را logoff می کند پوسته خاتمه پیدا می کند وسیستم، getty جدید راشروع می کند ومنتظر می ماند تا کاربر دیگری login شود.(ببینید در شکل ۴.۳ ) تشخیص اینکه پوسته فقط یک برنامه است مسئله مهمی است.واینکه پوسته هیچ امتیاز خاصی در سیستم ندارد به این معنا که هر کسی با تخصص واشتیاق کافی می تواند پوسته مورد نظر خود را بسازد. این دلیلی است بر اینکه امروزه انواع گوناگونی از پوسته ها به وجود آمده اند که پیشتر راجع به آنها سخن گفتیم.




شکل ۴.۳: چرخه login

## ۴.۳ مسئولیت های پوسته

اکنون ما می دانیم که پوسته هر خطی که کاربر تایپ می کند را تجزیه و تحلیل می کند وسپس اجرای برنامه منتخب آغاز می شود. اما پوسته وظایفی غیر از این را نیز داراست.(ببینید در شکل ۵.۳ )



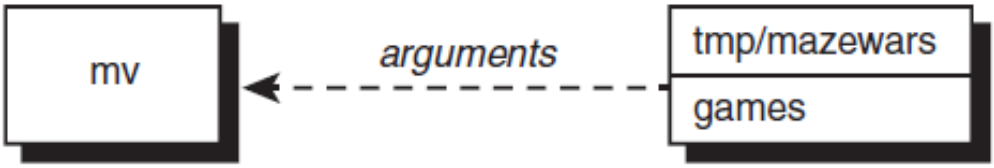

شکل ۵.۳: مسئولیت های پوسته

## ۵.۳ اجرای برنامه

پوسته مسئول اجرای تمام برنامه هایی است که توسط کاربر در ترمینال درخواست می شوند.هرگاه کاربر چیزی در یک خط برای پوسته تایپ می کند ،پوسته این خط را تجزیه وتحلیل می کند وتعیین می کند چه کاری باید انجام شود.هر خطی که کاربر می نویسد شکل پایه زیر را دنبال می کند : $program - name\ arguments$

یک خط کدی که برای پوسته تایپ می شود را " خط فرمان " می گویند. [۵] پوسته خط فرمان را اسکن می کند ونام برنامه ای که قرار است اجرا شود و آرگومان هایی که باید به برنامه فرستاده شود را تعیین می کند. پوسته از کراکترها ی خاصی برای تعیین شروع و پایان اسم برنامه وهمچنین آرگومان ها استفاده می کند که به این کرا کتر ها ،کراکترهای Whitespace می گویند. چندین رخداد ازکراکترهای whitespace توسط پوسته نادیده گرفته می شوند .مثلا اگر ما چند بار از کراکتر های whitespace پشت سرهم استفاده کنیم پوسته همه آنها را باهم یکبار به حساب می آورد.به طور مثال به نمونه زیر دقت کنید: $mv \quad tmp/mazewars\ games$

وقتی کاربر دستور بالا را وارد می کند پوسته این دستور که پیشتر گفته شد به آن خط فرمان می گوییم را اسکن می کند و از ابتدا تا رسیدن به اولین کراکتر whitespace را به عنوان اسم برنامه ای که قرار است اجرا شود در نظر می گیرد که در این نمونه اسم برنامه mv است. و بعد از این کراکتر whitespace ، کراکترهای whitespace بعدی (فضاهای خالی) در نظر گرفته نمی شوند تا به کراکتر بعدی که درواقع شروع آرگومان اول است برسیم بعد از آن تا رسیدن به فضای خالی بعدی به عنوان آرگومان اول که اینجا tmp/mazewars است در نظرگرفته می شود وبه همین ترتیب games به عنوان آرگومان دوم در نظر گرفته می شود. بعد از تجزیه وتحلیل این خط فرمان ، پوسته برای اجرای برنامه ای با نام mv وبا دو آرگومان tmp/mazewars وgames پیش می رود.( ببینید شکل ۶.۳ )




شکل ۶.۳: اجرای mv با دو آرگومان

## ۶.۳ جایگزینی متغیرو نام فایل

همانند بسیاری از زبان های برنامه نویسی پوسته نیز به کاربر این اجازه را می دهد که مقادیری را به متغیر ها اختصاص دهد. برای اختصاص دادن مقدار به متغیر در خط فرمان باید قبل اسم متغیر علامت $ را قرار بدهیم ،پوسته مقدار اختصاص داده شده به متغیر را جایگزین می کند.همچنین پوسته عمل جایگزینی نام فایل در خط فرمان را هم انجام می دهد.درحقیقت پوسته ،خط فرمان را اسکن کرده وبه دنبال کراکتر های جایگزینی فایل مثل *،؟و یا ]۰۰۰[ قبل از تعیین نام برنامه ای که قرار است اجرا شود وآرگومانهایش می گردد.

## ۷.۳ هدایت ورودی/خروجی

یکی دیگر از وظایف ومسئولیت های پوسته این است که ورودی / خروجی را بادقت هدایت کند. پوسته ، خط فرمان را برای رخدادهایی از کراکترهای هدایت > ، < ، « اسکن می کند. وقتی کاربر دستور

$$echo\ Remember\ to\ record\ The\ Walking\ Dead > reminder$$

را تایپ می کند ، پوسته کراکتر هدایت خروجی " < " را تشخیص داده وکلمه جدید در خط فرمان را به نام فایلی که خروجی باید به داخل آن هدایت شود می برد. در این نمونه نام فایل، reminder است .اگر reminder از قبل وجود داشته وکاربر دسترسی نوشتن داشته باشد آنگاه محتویات قبلی بازنویسی می شوند. اگر کاربر دسترسی نوشتن در فایل یا دایرکتوری اش نداشته باشد پوسته پیغام خطا می دهد. قبل از اینکه پوسته شروع به اجرای برنامه دلخواه کند ، ابتدا خروجی استاندارد را از برنامه به فایل نشان داده شده هدایت می کند. [۵] تقریبا در همه موارد برنامه هرگز نمی داند که خروجی اش به فایل نشان داده شده هدایت می شود. وفقط مسیر خود را برای نوشتن خروجی استاندارد طی می کند غافل از اینکه پوسته اطلاعات را به یک فایل هدایت کرده است.

## ۸.۳ کنترل محیط

پوسته فرمان های خاصی را فراهم می کند که به کاربر این اجازه را می دهند که محیط خود را خصوصی سازی کند . [۵] محیط کاربر*users home directory*، کراکتر هایی که پوسته برای سریع تر شدن تایپ کاربر نمایش می دهد و لیستی از دایرکتوری هایی که به هنگام اجرای برنامه مورد نظر کاربر جستجو می شوند را شامل می شود.

وبسیاری از مطالب دیگر دراین باره که در این مقاله به آن پرداخته نمی شود. در فصل بعد کد منبع مربوط به دو پوسته را می آوریم.

# فصل ۴

# کدهای منبع

در این فصل کد منبع مربوط به پوسته لینوکس وپوسته بورن را می آوریم.

## ۱.۴ کد منبع پوسته بورن

```
#define         INTR    2
#define         QUIT    3
#define LINSIZ  1000
#define ARGSIZ  50
#define TRESIZ  100

#define QUOTE  0200
#define FAND  1
#define FCAT  2
#define FPIN  4
#define FPOU  8
#define FPAR  16
#define FINT  32
#define FPRS  64
#define TCOM  1
#define TPAR  2
#define TFIL  3
```

```
#define TLST 4
#define DTYP 0
#define DLEF 1
#define DRIT 2
#define DFLG 3
#define DSPR 4
#define DCOM 5
#define         ENOMEM  12
#define         ENOEXEC 8

char    *dolp;
char    pidp[6];
int     ldivr;
char    **dolv;
int     dolc;
char    *promp;
char    *linep;
char    *elinep;
char    **argp;
char    **eargp;
int     *treep;
int     *treeend;
char    peekc;
char    gflg;
char    error;
char    acctf;
char    uid;
char    setintr;
char    *arginp;
int     onelflg;

char    *mesg[] {
0,
"Hangup",
0,
"Quit",
"Illegal␣instruction",
"Trace/BPT␣trap",
```

```
"IOT trap",
"EMT trap",
"Floating exception",
"Killed",
"Bus error",
"Memory fault",
"Bad system call",
0,
"Sig 14",
"Sig 15",
"Sig 16",
"Sig 17",
"Sig 18",
"Sig 19",
};

struct stime {
int proct[2];
int cputim[2];
int systim[2];
} timeb;

char  line[LINSIZ];
char  *args[ARGSIZ];
int   trebuf[TRESIZ];

main(c, av)
int c;
char **av;
{
register f;
register char *acname, **v;

for(f=2; f<15; f++)
close(f);
if((f=dup(1)) != 2)
close(f);
dolc = getpid();
```

```
for(f=4; f>=0; f--) {
dolc = ldiv(0, dolc, 10);
pidp[f] = ldivr+'0';
}
v = av;
acname = "/usr/adm/sha";
promp = "% ";
if(((uid = getuid())&0377) == 0)
promp = "# ";
acctf = open(acname, 1);
if(c > 1) {
promp = 0;
if (*v[1]=='-') {
**v = '-';
if (v[1][1]=='c' && c>2)
arginp = v[2];
else if (v[1][1]=='t')
onelflg = 2;
} else {
close(0);
f = open(v[1], 0);
if(f < 0) {
prs(v[1]);
err(": cannot open");
}
}
}
if(**v == '-') {
setintr++;
signal(QUIT, 1);
signal(INTR, 1);
}
dolv = v+1;
dolc = c-1;

loop:
if(promp != 0)
prs(promp);
```

```
peekc = getc();
main1();
goto loop;
}

main1()
{
register char c, *cp;
register *t;

argp = args;
eargp = args+ARGSIZ-5;
linep = line;
elinep = line+LINSIZ-5;
error = 0;
gflg = 0;
do {
cp = linep;
word();
} while(*cp != '\n');
treep = trebuf;
treeend = &trebuf[TRESIZ];
if(gflg == 0) {
if(error == 0) {
setexit();
if (error)
return;
t = syntax(args, argp);
}
if(error != 0)
err("syntax␣error"); else
execute(t);
}
}

word()
{
register char c, c1;
```

```
*argp++ = linep;

loop:
switch(c = getc()) {

case '␣':
case '\t':
goto loop;

case '\'':
case '"':
c1 = c;
while((c=readc()) != c1) {
if(c == '\n') {
error++;
peekc = c;
return;
}
*linep++ = c|QUOTE;
}
goto pack;

case '&':
case ';':
case '<':
case '>':
case '(':
case ')':
case '|':
case '^':
case '\n':
*linep++ = c;
*linep++ = '\0';
return;
}

peekc = c;
```

```
pack:
for(;;) {
c = getc();
if(any(c, "␣'\"\t;&<>()|^\n")) {
peekc = c;
if(any(c, "\"'"))
goto loop;
*linep++ = '\0';
return;
}
*linep++ = c;
}
}

tree(n)
int n;
{
register *t;

t = treep;
treep =+ n;
if (treep>treeend) {
prs("Command␣line␣overflow\n");
error++;
reset();
}
return(t);
}

getc()
{
register char c;

if(peekc) {
c = peekc;
peekc = 0;
return(c);
```

```
}
if(argp > eargp) {
argp =- 10;
while((c=getc()) != '\n');
argp =+ 10;
err("Too␣many␣args");
gflg++;
return(c);
}
if(linep > elinep) {
linep =- 10;
while((c=getc()) != '\n');
linep =+ 10;
err("Too␣many␣characters");
gflg++;
return(c);
}
getd:
if(dolp) {
c = *dolp++;
if(c != '\0')
return(c);
dolp = 0;
}
c = readc();
if(c == '\\') {
c = readc();
if(c == '\n')
return('␣');
return(c|QUOTE);
}
if(c == '$') {
c = readc();
if(c>='0' && c<='9') {
if(c-'0' < dolc)
dolp = dolv[c-'0'];
goto getd;
}
```

```
if(c == '$') {
dolp = pidp;
goto getd;
}
}
return(c&0177);
}

readc()
{
char cc;
register c;

if (arginp) {
if (arginp == 1)
exit();
if ((c = *arginp++) == 0) {
arginp = 1;
c = '\n';
}
return(c);
}
if (onelflg==1)
exit();
if(read(0, &cc, 1) != 1)
exit();
if (cc=='\n' && onelflg)
onelflg--;
return(cc);
}

/*
* syntax
*      empty
*      syn1
*/

syntax(p1, p2)
```

```
char **p1, **p2;
{

while(p1 != p2) {
if(any(**p1, ";&\n"))
p1++; else
return(syn1(p1, p2));
}
return(0);
}

/*
* syn1
*       syn2
*       syn2 & syntax
*       syn2 ; syntax
*/

syn1(p1, p2)
char **p1, **p2;
{
register char **p;
register *t, *t1;
int l;

l = 0;
for(p=p1; p!=p2; p++)
switch(**p) {

case '(':
l++;
continue;

case ')':
l--;
if(l < 0)
error++;
continue;
```

```
case '&':
case ';':
case '\n':
if(l == 0) {
l = **p;
t = tree(4);
t[DTYP] = TLST;
t[DLEF] = syn2(p1, p);
t[DFLG] = 0;
if(l == '&') {
t1 = t[DLEF];
t1[DFLG] =| FAND|FPRS|FINT;
}
t[DRIT] = syntax(p+1, p2);
return(t);
}
}
if(l == 0)
return(syn2(p1, p2));
error++;
}

/*
* syn2
*       syn3
*       syn3 | syn2
*/

syn2(p1, p2)
char **p1, **p2;
{
register char **p;
register int l, *t;

l = 0;
for(p=p1; p!=p2; p++)
switch(**p) {
```

```
case '(':
l++;
continue;

case ')':
l--;
continue;

case '|':
case '^':
if(l == 0) {
t = tree(4);
t[DTYP] = TFIL;
t[DLEF] = syn3(p1, p);
t[DRIT] = syn2(p+1, p2);
t[DFLG] = 0;
return(t);
}
}
return(syn3(p1, p2));
}

/*
* syn3
*      ( syn1 ) [ < in  ] [ > out ]
*      word word* [ < in ] [ > out ]
*/

syn3(p1, p2)
char **p1, **p2;
{
register char **p;
char **lp, **rp;
register *t;
int n, l, i, o, c, flg;

flg = 0;
```

```
if(**p2 == ')')
flg =| FPAR;
lp = 0;
rp = 0;
i = 0;
o = 0;
n = 0;
l = 0;
for(p=p1; p!=p2; p++)
switch(c = **p) {

case '(':
if(l == 0) {
if(lp != 0)
error++;
lp = p+1;
}
l++;
continue;

case ')':
l--;
if(l == 0)
rp = p;
continue;

case '>':
p++;
if(p!=p2 && **p=='>')
flg =| FCAT; else
p--;

case '<':
if(l == 0) {
p++;
if(p == p2) {
error++;
p--;
```

```
}
if(any(**p, "<>("))
error++;
if(c == '<') {
if(i != 0)
error++;
i = *p;
continue;
}
if(o != 0)
error++;
o = *p;
}
continue;

default:
if(l == 0)
p1[n++] = *p;
}
if(lp != 0) {
if(n != 0)
error++;
t = tree(5);
t[DTYP] = TPAR;
t[DSPR] = syn1(lp, rp);
goto out;
}
if(n == 0)
error++;
p1[n++] = 0;
t = tree(n+5);
t[DTYP] = TCOM;
for(l=0; l<n; l++)
t[l+DCOM] = p1[l];
out:
t[DFLG] = flg;
t[DLEF] = i;
t[DRIT] = o;
```

```
return(t);
}

scan(at, f)
int *at;
int (*f)();
{
register char *p, c;
register *t;

t = at+DCOM;
while(p = *t++)
while(c = *p)
*p++ = (*f)(c);
}

tglob(c)
int c;
{

if(any(c, "[?*"))
gflg = 1;
return(c);
}

trim(c)
int c;
{

return(c&0177);
}

execute(t, pf1, pf2)
int *t, *pf1, *pf2;
{
int i, f, pv[2];
register *t1;
register char *cp1, *cp2;
```

```
extern errno;

if(t != 0)
switch(t[DTYP]) {

case TCOM:
cp1 = t[DCOM];
if(equal(cp1, "chdir")) {
if(t[DCOM+1] != 0) {
if(chdir(t[DCOM+1]) < 0)
err("chdir: bad directory");
} else
err("chdir: arg count");
return;
}
if(equal(cp1, "shift")) {
if(dolc < 1) {
prs("shift: no args\n");
return;
}
dolv[1] = dolv[0];
dolv++;
dolc--;
return;
}
if(equal(cp1, "login")) {
if(promp != 0) {
close(acctf);
execv("/bin/login", t+DCOM);
}
prs("login: cannot execute\n");
return;
}
if(equal(cp1, "newgrp")) {
if(promp != 0) {
close(acctf);
execv("/bin/newgrp", t+DCOM);
}
```

```
prs("newgrp: cannot execute\n");
return;
}
if(equal(cp1, "wait")) {
pwait(-1, 0);
return;
}
if(equal(cp1, ":"))
return;

case TPAR:
f = t[DFLG];
i = 0;
if((f&FPAR) == 0)
i = fork();
if(i == -1) {
err("try again");
return;
}
if(i != 0) {
if((f&FPIN) != 0) {
close(pf1[0]);
close(pf1[1]);
}
if((f&FPRS) != 0) {
prn(i);
prs("\n");
}
if((f&FAND) != 0)
return;
if((f&FPOU) == 0)
pwait(i, t);
return;
}
if(t[DLEF] != 0) {
close(0);
i = open(t[DLEF], 0);
if(i < 0) {
```

```
prs(t[DLEF]);
err(": cannot open");
exit();
}
}
if(t[DRIT] != 0) {
if((f&FCAT) != 0) {
i = open(t[DRIT], 1);
if(i >= 0) {
seek(i, 0, 2);
goto f1;
}
}
i = creat(t[DRIT], 0666);
if(i < 0) {
prs(t[DRIT]);
err(": cannot create");
exit();
}
f1:
close(1);
dup(i);
close(i);
}
if((f&FPIN) != 0) {
close(0);
dup(pf1[0]);
close(pf1[0]);
close(pf1[1]);
}
if((f&FPOU) != 0) {
close(1);
dup(pf2[1]);
close(pf2[0]);
close(pf2[1]);
}
if((f&FINT)!=0 && t[DLEF]==0 && (f&FPIN)==0) {
close(0);
```

```
open("/dev/null", 0);
}
if((f&FINT) == 0 && setintr) {
signal(INTR, 0);
signal(QUIT, 0);
}
if(t[DTYP] == TPAR) {
if(t1 = t[DSPR])
t1[DFLG] =| f&FINT;
execute(t1);
exit();
}
close(acctf);
gflg = 0;
scan(t, &tglob);
if(gflg) {
t[DSPR] = "/etc/glob";
execv(t[DSPR], t+DSPR);
prs("glob: cannot execute\n");
exit();
}
scan(t, &trim);
*linep = 0;
texec(t[DCOM], t);
cp1 = linep;
cp2 = "/usr/bin/";
while(*cp1 = *cp2++)
cp1++;
cp2 = t[DCOM];
while(*cp1++ = *cp2++);
texec(linep+4, t);
texec(linep, t);
prs(t[DCOM]);
err(": not found");
exit();

case TFIL:
f = t[DFLG];
```

```
pipe(pv);
t1 = t[DLEF];
t1[DFLG] =| FPOU | (f&(FPIN|FINT|FPRS));
execute(t1, pf1, pv);
t1 = t[DRIT];
t1[DFLG] =| FPIN | (f&(FPOU|FINT|FAND|FPRS));
execute(t1, pv, pf2);
return;

case TLST:
f = t[DFLG]&FINT;
if(t1 = t[DLEF])
t1[DFLG] =| f;
execute(t1);
if(t1 = t[DRIT])
t1[DFLG] =| f;
execute(t1);
return;

}
}

texec(f, at)
int *at;
{
extern errno;
register int *t;

t = at;
execv(f, t+DCOM);
if (errno==ENOEXEC) {
if (*linep)
t[DCOM] = linep;
t[DSPR] = "/bin/sh";
execv(t[DSPR], t+DSPR);
prs("No␣shell!\n");
exit();
}
```

```
if (errno==ENOMEM) {
prs(t[DCOM]);
err(": too large");
exit();
}
}

err(s)
char *s;
{

prs(s);
prs("\n");
if(promp == 0) {
seek(0, 0, 2);
exit();
}
}

prs(as)
char *as;
{
register char *s;

s = as;
while(*s)
putc(*s++);
}

putc(c)
{

write(2, &c, 1);
}

prn(n)
int n;
{
```

```
register a;

if(a=ldiv(0,n,10))
prn(a);
putc(lrem(0,n,10)+'0');
}

any(c, as)
int c;
char *as;
{
register char *s;

s = as;
while(*s)
if(*s++ == c)
return(1);
return(0);
}

equal(as1, as2)
char *as1, *as2;
{
register char *s1, *s2;

s1 = as1;
s2 = as2;
while(*s1++ == *s2)
if(*s2++ == '\0')
return(1);
return(0);
}

pwait(i, t)
int i, *t;
{
register p, e;
int s;
```

```
if(i != 0)
for(;;) {
times(&timeb);
time(timeb.proct);
p = wait(&s);
if(p == -1)
break;
e = s&0177;
if(mesg[e] != 0) {
if(p != i) {
prn(p);
prs(": ");
}
prs(mesg[e]);
if(s&0200)
prs(" -- Core dumped");
}
if(e != 0)
err("");
if(i == p) {
acct(t);
break;
} else
acct(0);
}
}

acct(t)
int *t;
{
if(t == 0)
enacct("**gok"); else
if(*t == TPAR)
enacct("()"); else
enacct(t[DCOM]);
}
```

```
enacct(as)
char *as;
{
struct stime timbuf;
struct {
char cname[14];
char shf;
char uid;
int datet[2];
int realt[2];
int bcput[2];
int bsyst[2];
} tbuf;
register i;
register char *np, *s;

s = as;
times(&timbuf);
time(timbuf.proct);
lsub(tbuf.realt, timbuf.proct, timeb.proct);
lsub(tbuf.bcput, timbuf.cputim, timeb.cputim);
lsub(tbuf.bsyst, timbuf.systim, timeb.systim);
do {
np = s;
while (*s != '\0' && *s != '/')
s++;
} while (*s++ != '\0');
for (i=0; i<14; i++) {
tbuf.cname[i] = *np;
if (*np)
np++;
}
tbuf.datet[0] = timbuf.proct[0];
tbuf.datet[1] = timbuf.proct[1];
tbuf.uid = uid;
tbuf.shf = 0;
if (promp==0)
tbuf.shf = 1;
```

```
  seek(acctf, 0, 2);
  write(acctf, &tbuf, sizeof(tbuf));
  }
```

## ۲.۴ کد منبع پوسته لینوکس

```
// Shell start file


#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>

#define COMMAND_LENGTH 1024
#define NUM_TOKENS (COMMAND_LENGTH / 2 + 1)
#define HISTORY_DEPTH 10


/**
* Command Input and Processing
*/
int index = 0;
int counter = 0;
int historyIterator = 0;
char temp_buffer[HISTORY_DEPTH][COMMAND_LENGTH];

void collect_history(char *buff)
{
char shift_buffer[HISTORY_DEPTH-1][COMMAND_LENGTH];
```

```
if (index >= 10)
{
for(int shift = 0; shift < 9; shift++)
{
memmove(shift_buffer[shift], temp_buffer[shift+1],
strlen(temp_buffer[shift+1]));
memset(temp_buffer[shift+1], 0,
strlen(temp_buffer[shift+1]));
}
memset(temp_buffer[0], 0,
strlen(temp_buffer[0]));
for(int shift = 0; shift < 9; shift++)
{
memmove(temp_buffer[shift], shift_buffer[shift],
strlen(shift_buffer[shift]));
memset(shift_buffer[shift], 0,
strlen(shift_buffer[shift]));
}
memset(temp_buffer[9], 0, strlen(temp_buffer[9]));
memmove(temp_buffer[9], buff, strlen(buff));
counter = 0;
}
Else
{
memmove(temp_buffer[counter], buff, strlen(buff));
counter++;
}
index++;
}

int indexCall(char *buff)
{
char maxIndex[COMMAND_LENGTH];
char minIndex[COMMAND_LENGTH];
char check[COMMAND_LENGTH];
sprintf(maxIndex, "%i", index);
sprintf(minIndex, "%i", index-10);
```

```
if (index != 0)
{
if (buff[1] != '!')
{
/*
*
*       Handles the command !n
*
*/
if ((strlen(buff)-1 == strlen(maxIndex)) ||
((strlen(buff)-1 == strlen(minIndex))))
{
int x = 0;
int numberCap = 0;
if (index <= 10)
{
numberCap = index;
}
Else
{
numberCap = 10;
}

for(int i = 0; i <= numberCap; i++)
{
if (buff[i+1] <= maxIndex[i])
{
x++;
}
else
{
x--;
}

if (buff[i+1] >= minIndex[i])
{
x++;
}
```

```
Else
{
x--;
}
}
if (x < 0)
{
write(STDOUT_FILENO, "SHELL: Unknown history    command.",
strlen("SHELL: Unknown history command."));
historyIterator--;
return 1;
}

if (x >= 0)
{
buff[0] = '\n';
memmove(buff, buff+1, strlen(buff));
int convertedIndex = -1;
int y = index - 10;
if (index <= 10)
{
y = 0;
}
sprintf(check, "%i", y);
while(memcmp(buff, check, strlen(check)) != 0)
{
y++;
convertedIndex++;
sprintf(check, "%i", y);

}
memset(buff, 0 , strlen(buff));
memmove(buff, temp_buffer[convertedIndex],
strlen(temp_buffer[convertedIndex]));
}

}
Else
```

```
{
write(STDOUT_FILENO, "SHELL: Unknown history command.",
strlen("SHELL: Unknown history command."));
historyIterator--;
return 1;
}
}
/*
*
*        Handles the command !!
*
*/
if (buff[1] == '!')
{
if (index <= 10)
{
memset(buff, 0 , strlen(buff));
memmove(buff, temp_buffer[index-1], strlen(temp_buffer[index-1]));
}
Else
{
memset(buff, 0 , strlen(buff));
memmove(buff, temp_buffer[9], strlen(temp_buffer[9]));
}
}

}
if (index == 0)
{
write(STDOUT_FILENO, "SHELL: Unknown history command.",
strlen("SHELL: Unknown history command."));
historyIterator--;
return 1;
}
return 0;
}
```

```
/*
* Tokenize the string in 'buff' into 'tokens'.
* buff: Character array containing string to tokenize.
* Will be modified: all whitespace replaced with '\0'
* tokens: array of pointers of size at least
COMMAND_LENGTH/2 + 1.
* Will be modified so tokens[i] points to the i'th token
* in the string buff. All returned tokens will be non-empty.
* NOTE: pointers in tokens[] will all point into buff!
* Ends with a null pointer.
* returns: number of tokens.
*/
int tokenize_command(char *buff, char *tokens[])
{
int addHistory = 0;
if ((buff[0] == '!'))
{
addHistory = indexCall(buff);
}
if((buff[0]) == '\0')
{
addHistory = 1;
historyIterator--;
}

if (addHistory != 1)
{
collect_history(buff);
}

/*if (buff[0] == '!')
{
collect_exclamation(buff);
}*/
int token_count = 0;
_Bool in_token = false;
int num_chars = strnlen(buff, COMMAND_LENGTH);
```

```
for (int i = 0; i < num_chars; i++) {
switch (buff[i]) {
// Handle token delimiters (ends):
case ' ':
case '\t':
case '\n':
buff[i] = '\0';
in_token = false;
break;

// Handle other characters (may be start)
default:
if (!in_token) {
tokens[token_count] = &buff[i];
token_count++;
in_token = true;
}
}
}
tokens[token_count] = NULL;
return token_count;
}

/**
* Read a command from the keyboard into the buffer 'buff' and
tokenize it
* such that 'tokens[i]' points into 'buff' to the i'th
token in the command.
* buff: Buffer allocated by the calling code.
Must be at least
* COMMAND_LENGTH bytes long.
* tokens[]: Array of character pointers which point into
'buff'. Must be at
* least NUM_TOKENS long. Will strip out up to one final
```

```
'&' token.
* tokens will be NULL terminated (a NULL pointer
indicates end of tokens).
* in_background: pointer to a boolean variable.
Set to true if user entered
* an & as their last token; otherwise set to false.
*/
void read_command(char *buff, char *tokens[],
_Bool *in_background)
{
*in_background = false;

// Read input

int length = read(STDIN_FILENO, buff, COMMAND_LENGTH-1);
if ( (length < 0) && (errno !=EINTR) )
{
perror("Unable to read command. Terminating.\n");
exit(-1); /* terminate with error */
}

// Null terminate and strip \n.
buff[length] = '\0';
if (buff[strlen(buff) - 1] == '\n') {
buff[strlen(buff) - 1] = '\0';
}


// Tokenize (saving original command string)
int token_count = tokenize_command(buff, tokens);
if (token_count == 0) {
return;
}

// Extract if running in background:
if (token_count > 0 && strcmp(tokens[token_count - 1], "&") == 0) {
*in_background = true;
tokens[token_count - 1] = 0;
```

```
}
}
void history()
{
if (index > 10)
{
historyIterator = 10;
}

int historyIndex = index - historyIterator + 1;
char historyIndexS[] = "0";

for(int i = 0; i < historyIterator; i++)
{
sprintf(historyIndexS, "%i", historyIndex);
write(STDOUT_FILENO, historyIndexS, strlen(historyIndexS));
write(STDOUT_FILENO, ":\t", strlen(":\t"));
write(STDOUT_FILENO, temp_buffer[i], sizeof(temp_buffer[i]));
write(STDOUT_FILENO, "\n", strlen("\n"));
historyIndex++;
}
}

void siginthandler(int param)
{
history();
signal(SIGINT, siginthandler);
}

/**
* Main and Execute Commands
*/
int main(int argc, char* argv[])
{
char input_buffer[COMMAND_LENGTH];
char *tokens[NUM_TOKENS];
char execvpError[] = "Unknown command.\n";
char invalidDir[] = "Invalid Directory.\n";
```

```
char currentPath[COMMAND_LENGTH];

while (true)
{
int status = 0;
int error = 0;
// Get command
// Use write because we need to use read() to work withmemcpy
// signals, and read() is incompatible with printf().
getcwd(currentPath, sizeof(currentPath));
write(STDOUT_FILENO, currentPath, strlen(currentPath));
write(STDOUT_FILENO, "> ", strlen("> "));
_Bool in_background = false;
signal(SIGINT, siginthandler);

read_command(input_buffer, tokens, &in_background);


/*
*
*       2.2 Internal Commands
*       2.3 Creating a History Feature
*
*/
historyIterator++;

if(tokens[0] != NULL)
{
if (strcmp(tokens[0], "pwd") == 0)
{
getcwd(currentPath, sizeof(currentPath));
write(STDOUT_FILENO, currentPath, strlen(currentPath));
status = 1;
}
else if (strcmp(tokens[0], "cd") == 0)
{
int checkDir = chdir(tokens[1]);
if (checkDir == -1)
```

```
{
write(STDOUT_FILENO, invalidDir, strlen(invalidDir));
}
status = 1;
}
else if (strcmp(tokens[0], "exit") == 0)
{
exit(0);
}
else if (strcmp(tokens[0], "history") == 0)
{
history();
status = 1;
}
else if (strncmp(tokens[0], "!", 1) == 0)
{
status = 1;
}
}

/*
*
*       End of 2.2 & 2.3
*
*/

if (status == 0)
{
pid_t pid = fork();
if (pid < 0)
{
perror("Failed to create child");
}
else if (pid == 0)
{
error = execvp(tokens[0], tokens);
if (error == (-1))
{
```

```
write(STDOUT_FILENO, tokens[0], strlen(tokens[0]));
write(STDOUT_FILENO, ":\t", strlen(":\t"));
write(STDOUT_FILENO, execvpError, sizeof(execvpError));
exit(0);
}
}
if(!in_background)
{
waitpid(pid, NULL, 0);
}
}


// DEBUG: Dump out arguments:
for (int i = 0; tokens[i] != NULL; i++) {
write(STDOUT_FILENO, "␣␣␣Token:␣", strlen("␣␣␣Token:␣"));
write(STDOUT_FILENO, tokens[i], strlen(tokens[i]));
write(STDOUT_FILENO, "\n", strlen("\n"));
}
if (in_background)
{
write(STDOUT_FILENO, "Run␣in␣background.",
strlen("Run␣in␣background."));
}

/**
* Steps For Basic Shell:
* 1. Fork a child process
* 2. Child process invokes execvp()
using results in token array.
* 3. If in_background is false, parent waits for
*    child to finish. Otherwise, parent loops back to
*    read_command() again immediately.
*/

}
return 0;
}
```

# کتاب‌نامه


[۱] یونیکس ـ ویکی پدیا، دانشنامهٔ آزاد

[۲] پوسته چیست؟$(What's\ shell)$

[۳] لیلا آقایی میبدی،ل:"بازبینی وتحلیل سیستم عامل یونیکس".

[4] $(What's\ shell).html$

[5] $Stephen\ G.\ Kochan, Patrick\ Wood - S, P(2017).$
$Shell\ Programming\ in\ Unix, Linux\ and\ OSX 4978-0-13-4449600-$
$9 Addison - Wesley The\ United\ States\ of\ Amrica.$

[6] $http://digispark.ir/evolution-unix-linux-shells/amp/$

**Abstract**

A Unix shell is a command-line interpreter or shell that provides a Traditional Unix-like command line user interface.In this project,We intend to express the precise concept of the Unix shell, the types of shells, as well as the importance of the shell.

College of Science
School of Mathematics, Statistics, and Computer Science

# Unix Shell

**Najme Dastorani**

Supervisor: Engineer Ebrahim Naghibzade Mashayekh

A thesis submitted to Graduate Studies Office
in partial fulfillment of the requirements for the degree of
B.Sc.Computer Science

Summer 2018