گزارش پروژه دوم آزمایشگاه سیستم عامل

پریا خوش‌تاب ۸۱۰۱۹۸۳۸۷

علی اخگری ۸۱۰۱۹۸۳۴۱

پرنیان فاضل ۸۱۰۱۹۸۵۱۶

مقدمه

۱) کتابخانه های (قاعدتاً سطح کاربر، منظور فایل‌های تشکیل دهنده متغیر ULIB در Makefile است) استفاده شده در xv6 را از منظر استفاده از فراخوانی‌های سیستمی و علت این استفاده بررسی نمایید.

فایل‌های تشکیل دهنده متغیر ULIB شامل printf.c، usys.s، ulib.c و umalloc.c می‌باشد. فایل های printf.c، ulib.c و umalloc.c شامل کتابخانه های سطح کاربر مانند free، malloc، printf و ... می باشد.

printf.c شامل توابع زیر می باشد:

- putc: این تابع یک کاراکتر را در مقصد که توسط یک file descriptor مشخص شده است، چاپ می‌کند. در این تابع از سیستم کال write استفاده شده است.
- printint: تابعی است که یک مقدار integer را در مقصد که توسط یک file descriptor مشخص شده است، چاپ می کند که برای این منظور تمام ارقام بافر را توسط تابع putc در مقصد چاپ می کند.

- printf: این تابع یک string شامل اعداد و کاراکتر ها را در مقصد چاپ می کند که برای این منظور از توابع putc و printint استفاده می کند.

:ulib.c

- stat: این تابع اطلاعات مربوط به یک فایل را در قالب ساختار stat بر میگرداند. در این تابع از سیستم کال های close، open و fstat استفاده می شود.
- شامل توابع atoi، strlen، strcmp، strcpy و ...می باشد که توابع مربوط به کار کردن با string ها می باشد.

:umalloc.c

- free: این تابع یک پوینتر را به عنوان ورودی می‌گیرد که این پوینتر به یک آدرس از حافظه اشاره می کند و سپس این بخش از حافظه را آزاد می کند.
- morecore: این تابع توسط سیستم کال sbrk، حافظه پردازه را گسترش می دهد.
- malloc: برای تخصیص حافظه ی پویا با اندازه ی مشخص استفاده می شود که مقدار خروجی آن یک اشاره گر void به حافظه ی تخصیص داده شده است.

usys.S: این فایل شامل پوشاننده های سیستم‌کال ها، به زبان اسمبلی می باشد. در این فایل یک تابع به نام SYSCALL(name) وجود دارد که ابتدا مقدار SYS_name را در رجیستر eax می ریزد و سپس یک وقفه با کد T_SYSCALL تولید می کند.

۲) دقت شود فراخوانی های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روشها را در لینوکس به اختصار توضیح دهید.

روش اول: از طریق file system

لینوکس دارای API های دیگری است که از طریق pseudo-file system ها مانند /proc/ ,/sys/ ,/dev/ به هسته دسترسی پیدا می کند. به طور مثال فایل سیستم /proc ارتباط بین فضای هسته و فضای کاربر را فراهم

می کند. /sys یک رابط برای هسته است که به طور مشخص تر، یک نمای سیستم فایلی از اطلاعات و تنظیمات پیکربندی هسته را ارائه می کند. /dev محل دیوایس فایل ها می باشد.

روش دوم: از طریق signal

یکی دیگر از روش های دسترسی از سطح کاربر به سطح هسته در لینوکس، استفاده از سیگنال ها است. سیگنال یک پیام بسیار کوتاه است که ممکن است به یک فرآیند یا گروهی از فرآیندها ارسال شود. تنها اطلاعاتی که به فرآیند داده می شود معمولاً عددی است که سیگنال را شناسایی می کند. سیگنال های استاندارد آرگومان ها یا سایر اطلاعات را منتقل نمی کنند. مجموعه ای از ماکروها که نام آنها با پیشوند SIG شروع می شود برای شناسایی سیگنال ها استفاده می شود.

سیگنال ها دو هدف اصلی را دنبال می کنند:

۱- برای آگاه ساختن یک فرآیند از وقوع یک رویداد خاص

۲- مجبور کردن یک فرآیند به اجرای handler function آن سیگنال

روش سوم: استفاده از socket

در این حالت برنامه هایی که در سطح کاربر هستند می توانند با استفاده از سوکت روی یک port خاص گوش دهند و اطلاعات را رد و بدل کنند.

ساز و کار اجرای فراخوانی سیستمی در xv6

۳) آیا باقی تله ها را نمی توان با سطح دسترسی DPL_USER فعال نمود؟ چرا؟

خیر، سطوح سیستم عامل از شماره ۰ تا ۳ نامگذاری شدند که در سیستم عامل xv6 تنها دو سطح ۰ یعنی سطح کرنل و سطح ۳ یعنی سطح کاربر وجود دارد. در فراخوانی های سیستمی، تله ها در سطح ۳ یعنی DPL_USER فعال می‌شوند زیرا فراخوانی‌های سیستمی توسط قطعه کدی در سطح کاربر فراخوانی می‌گردد و اگر برنامه سطح کاربر سطح دسترسی مناسبی نداشته باشد، دستور int باعث به وجود آمدن دستور int 13

می‌شود که یک خطای حفاظتی (protection) است. اما مابقی تله‌ها که از سطح کرنل صدا زده می‌شوند که عموما مربوط به درایور های سیستم و حافظه سیستم می‌باشند ، باید از سطح دسترسی ۰ فعال شوند.

۴) در صورت تغییر سطح دسترسی، ss و esp روی پشته Push میشود. در غیر این صورت Push نمی شود. چرا؟

به طور کلی دو پشته داریم که یکی برای سطح کاربر و دیگری برای سطح kernel است. هنگامی که می خواهیم سطح دسترسی را تغییر دهیم، باید ss و esp را روی پشته push کنیم تا هنگام بازگشت بدانیم که اخرین دستوری که انجام داده ایم چه بوده است و ادامه روند اجرای دستورات را از سر بگیریم. حال با توجه به اینکه داشتن این اطلاعات ضروری است، نیاز داریم که هر بار با تغییر سطح دسترسی این اطلاعات را روی پشته push کنیم. به همین ترتیب زمانی که تغییر سطح نداشته باشیم، نیاز نداریم که این اطلاعات را روی پشته push کنیم.

۵) در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در argptr() بازه آدرسها بررسی میگردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی بازه ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی sys_read() اجرای سیستم را با مشکل روبرو سازد.

از توابع دسترسی به توابع argptr() و argint() و argstr() و argfd() هستند که به ترتیب از argint() و fetchint() و fetchstr() در فایل syscall.c و argfd() در فایل sysfile.c استفاده می کند.

argint(): این تابع دو ورودی دارد. شماره پارامتر به عنوان ورودی اول و ورودی دوم به صورت pass by refernce که یک متغیر با int type است، داده می شود که در واقع مقدار پارامتر در این متغیر (ورودی دوم) ذخیره می شود. این تابع با صدا زدن تابع fetchinit() و دادن مقدار (myproc()->tf->esp) + 4 + 4*n به عنوان ورودی اول آن، آدرس پارامتر خواسته شده را ساخته و محتوای آن را برمیگرداند. توجه شود که اگر این

هدف با موفقیت انجام شود مقدار صفر و اگر این پارامتر وجود نداشته باشد و عملیات با موفقیت انجام نشود، خروجی ۱- برگردانده می شود.

argptr(): این تابع سه ورودی دارد. شماره پارامتر خواسته شده به عنوان ورودی اول و ورودی دوم به صورت pass by reference و سایز پارامتر به عنوان ورودی سوم به تابع داده می شود که ورودی دوم محتوای پارامتر های به شکل pointer مانند آرایه را در ورودی دوم ذخیره می کند. این تابع با صدا زدن تابع argint() آدرس خانه اول این پارامتر از جنس اشاره گر را بر می گرداند. توجه شود که این تابع در صورتی که با موفقیت عملیات را انجام دهد مقدار صفر و در صورت شکست مقدار ۱- را بر می گرداند.

argstr(): این تابع دو ورودی دارد. شماره پارامتر به عنوان ورودی اول و ورودی دوم به صورت pass by refernce به عنوان ورودی دوم به تابع داده می شود. این تابع برای مشخص کردن پارامتر های از جنس رشته است. این تابع ابتدا با استفاده از argint() آدرس خانه اول رشته را به دست می آورد و سپس با صدا زدن تابع fetchstr() رشته را در ورودی دوم این تابع ذخیره می کند. توجه شود که این تابع در صورتی که با موفقیت عملیات را انجام دهد مقدار صفر و در صورت شکست مقدار ۱- را بر می گرداند.

argfd(): این تابع در فایل sysfile.c تعریف شده و سه ورودی دارد.ورودی اول شماره پارامتر است. ورودی دوم آن اشاره گری به توصیف کننده فایل یا همان file descriptor و ورودی سوم آن اشاره گری به آدرس ساختمان داده ی file است که پس از انجام عملیات تابع مقدار دهی می شوند. توجه شود که این تابع در صورتی که با موفقیت عملیات را انجام دهد مقدار صفر و در صورت شکست مقدار ۱- را بر می گرداند.

در تابع argptr() پس از مشخص شدن آدرس شروع توسط argint() اگر این تابع ۱- برگرداند یعنی آدرس شروع نامعتبر بوده و عملیات با شکست مواجه شده. سپس سه شرط چک می شود: ۱- سایز منفی نباشد ۲- آدرس شروع از سایز پردازه فعلی کمتر نباشد ۳- جمع آدرس شروع و سایز داده شده در ورودی سوم بیشتر از سایز این پردازه باشد. در صورتی که یکی از ۳ شرط برقرار باشد یعنی پارامتر معتبر نبوده و دسترسی ما به جای اشتباهی از استک صورت گرفته و اشاره گر به جایی غیر از قسمت حافظه اختصاص داده شده به این پردازه اشاره دارد پس

عملیات با شکست مواجه شده و ۱- برگردانده می شود. اگر این بازه ها چک نشوند ممکن است ما محتوای قسمت هایی از حافظه را بخوانیم که اشتباه هستند و بنابراین برنامه اشتباه انجام میشود و یا ممکن است به قسمتی از حافظه دسترسی پیدا کنیم و تغییری در آن ایجاد کنیم که در این صورت ممکن است باعث بروز خطا در پردازه های دیگر و یا افتادن در تله شویم.

در فراخوانی سیستمی ()sys_read اگر این بازه ها چک نشوند برای مثال ممکن است محتوای فایلی که میخوانیم در قسمت نادرستی ذخیره شود که مخصوص این پردازه نیست و بنابراین هم در انجام عملیات پردازه فعلی مشکل ایجاد می شود و هم ممکن است در ایجاد عملیات دیگر پردازه ها ایجاد مشکل وند و بخشی از اطلاعات حافظه از دست برود و یا در تله بیفتیم.

۶- چگونه هنگام بازگشت به سطح کاربر، اجرا از همان خطی که متوقف شده بود، دوباره از سر گرفته میشود؟ فرایند را توضیح دهید.

روند به این صورت است که هنگامی که یک interrupt صورت می گیرد، روند عادی اجرای پردازه ها متوقف می شود و یک سری دیگر از کار ها شروع به انجام شدن می کنند که به این سری از کار ها interrupt handler می گویند.

قبل از اینکه interrupt handler شروع به کار کند، پردازه رجیستر های مربوط به این interrupt را ذخیره می کند که در نتیجه سیستم عامل وقتی که از interrupt باز می گردیم، می تواند این مقادیر را بازگردانی کند.

برای مثال وقتی که دستور وقفه (int n) اجرا می‌شود، ابتدا nامین descriptor از IDT، fetch می‌شود و سپس چک می‌شود که مقدار CPL که در رجیستر %cs ذخیره شده از DPL بزرگ‌تر نباشد یعنی CPL<=DPL و در صورتی که این شرط برقرار بود، مقادیر رجیسترهای %ss و %esp در رجیسترهای داخلی پردازنده ذخیره می‌شود و پس از لود شدن مقدار این متغیرها از task segment descriptor، این رجیسترها به ترتیب روی استک push می‌شوند و سپس به ترتیب رجیسترهای %eflag، %cs، %eip روی استک push می‌شوند که رجیستر %eip اشاره گر دستور می باشد و در واقع آدرس بازگشت در سطح کاربر در آن ذخیره می‌شود.

ارسال آرگومان های فراخوانی های سیستمی

برای اضافه کردن یک سیستم کال، باید مراحل زیر انجام شود:

۱- در فایل syscall.h یک عدد برای سیستم کال های جدیدمان تعریف کنیم.

```
24  #define SYS_calculate_sum_of_digits 22
25  #define SYS_get_file_sectors 23
26  #define SYS_get_parent_pid 24
27  #define SYS_set_process_parent 25
```

۲- در فایل prototype ،syscall.c توابع این سیستم کال ها را اضافه می کنیم.

```
107  extern int sys_calculate_sum_of_digits(void);
108  extern int sys_get_file_sectors(void);
109  extern int sys_get_parent_pid(void);
110  extern int sys_set_process_parent(void);
```

۳- در فایل syscall.c یک پوینتر به سیستم کال ها اضافه می کنیم. در این فایل یک آرایه از pointer function ها داریم که به واسطه ی اعداد نسبت داده شده به سیستم کال ها، یک پوینتر به سیستم کال ها تعریف می کند. با این کار هر زمان که یک سیستم کال را با شماره متناظر آن صدا کنیم، تابعی که آن سیستم کال به آن اشاره می کند را صدا خواهیم کرد.

```
135  [SYS_calculate_sum_of_digits]  sys_calculate_sum_of_digits,
136  [SYS_get_file_sectors]  sys_get_file_sectors,
137  [SYS_get_parent_pid]  sys_get_parent_pid,
138  [SYS_set_process_parent]  sys_set_process_parent,
```

۴- حال باید تابع سیستم کال ها را تعریف کنیم. این کار را بسته به اینکه چه سیستم کالی داریم، می توان در دو فایل sysproc.c و sysfile.c انجام داد.

۵- در فایل usys.S باید بگوییم که عدد سیستم کال هایی را که اضافه کردیم را در رجیستر eax بریزد. با این کار، توسط دستور int $T_SYSCALL می توان به سیستم عامل رخ دادن آن سیستم کال را اعلام کرد.

```
32  SYSCALL(calculate_sum_of_digits)
33  SYSCALL(get_file_sectors)
34  SYSCALL(get_parent_pid)
35  SYSCALL(set_process_parent)
```

۶- در فایل user.h باید prototype توابع را بنویسیم که برنامه سطح کاربر به واسطه صدا کردن آن ها، به سیستم کامل مورد نظر متصل می شود.

```
int calculate_sum_of_digits(void);
int get_file_sectors(int, int *sectors);
int get_parent_pid(void);
void set_process_parent(int);
```

۷- در انتها باید یک فایل اضافه کنیم که در آن فایل، کار مورد نظر را با صدا کردن تابعی که در user.h است، انجام دهیم. همچنین باید در Makefile، در قسمت UPROGS که دستورات وجود دارند، دستورات مورد نظر را اضافه می کنیم و همینطور در قسمت EXTRA باید فایل های مورد نظر را اضافه کنیم.

۱- پیاده سازی فراخوانی سیستمی(int n)calculate_sum_of_digits:

در این سیستم کال باید جمع ارقام را حساب کنیم و آن را خروجی دهیم. در این سیستم کال باید به جای بازیابی آرگومان ها به روش معمول، از رجیستر ها استفاده کنیم.

ابتدا مراحل اولیه ذکر شده در بالا را برای اضافه کردن فراخوانی سیستمی انجام می دهیم. در فایل سطح کاربر ابتدا پارامتر n یعنی argv[1] را با استفاده از دستور atoi() به integer تبدیل می کنیم و سپس برای ذخیره کردن این عدد در رجیستر(ebx) از کد اسمبلی استفاده می کنیم. در کد اسمبلی برای این که مقدار اولیه رجیستر ذخیره شود و از بین نرود و در آخر برنامه رجیستر مورد نظر همان مقدار قبلی را داشته باشد، در کد اسمبلی پس از ذخیره کردن n در رجیستر ebx، مقدار ذخیره شده در این رجیستر را در یک متغیر به نام saved_ebx ذخیره می کنیم. سپس تابع calculate_sum_of_digits() را صدا می زنیم و مقدار بازگشتی آن را چاپ میکنیم و در نهایت در فایل سطح کاربر دوباره با استفاده از کد اسمبلی مقدار متغیر saved_ebx را دوباره در رجیستر ebx ذخیره می کنیم.

حال تابع فراخوانی سیستمی مورد نظر را در فایل sysproc.c اضافه می کنیم. در این تابع ابتدا عددی را که در رجیستر ebx ذخیره کردیم را با استفاده از trapframe پردازه کنونی به دست می آوریم و سپس برای محاسبه مجموع ارقام این عدد هربار باقیمانده این عدد بر 10 را باهم جمع می کنیم و عدد را تقسیم بر 10 میکنیم تا زمانی که این عدد از 0 بزرگ تر باشد و در نهایت مجموع این ارقام را برمیگردانیم.

۲- پیاده سازی فراخوانی سیستمی پیدا کردن آدرس سکتورهای فایل:

get_file_sectors(int fd, int *sector)

در این بخش ما داده ساختارهای شرح داده شده در صورت پروژه را در xv6 پیدا کردیم و جزییات آن را مطالعه کردیم و فراخوانی سیستمی گفته شده را طراحی کردیم که بعد از باز کردن فایل، توصیف کننده فایل را به عنوان ورودی دریافت نموده و آدرس سکتورهای فایل را به سطح کاربر برگردانده و چاپ می کند. برای این کار در فایل int sys_get_file_sectors(void) با استفاده از argfd() استراکت file را

طبق پارامتر اول را پر می کنیم. سپس از طریق این استراکت می توانیم به ip* struct inode دسترسی پیدا کنیم و از این طریق می توانیم با استفاده از آرایه addrs در استراکت inode به آدرس های سکتور ها دسترسی پیدا کنیم و در نهایت با تابع bmap در فایل fs.c آدرس سکتور ها را پیدا کنیم. توجه شود که چون تابع bmap در فایل fs.c به صورت static تعریف شده است باید یک تابع دیگر در fs.c تعریف کنیم تا مقدار bmap را برگردانده اما static نباشد تا بتوانیم از sysfile.c به فایل fs.c دسترسی داشته باشیم. این تابع را به نام get_bmap در فایل fs.c تعریف کردیم. همچنین امضای این تابع را باید در فایل defs.h اضافه کنیم. حال برای به دست آوردن آدرس ها در تابع get_file_sectors به این صورت عمل می کنیم: پس از اینکه استراکت file ساخته شد، از طریق تابع ()argptrint که خودمان تعریف کردیم، به آرایه sectors که ورودی دوم تابع get_file_sectors است، دسترسی پیدا می کنیم تا بتوانیم آن را با آدرس های سکتور ها پر کنیم. این تابع را بعدا توضیح می‌دهیم. در تابع get_file_sectors پس از خواندن ۲ پارامتر توسط argfd و argptrint یک حلقه while میزنیم و با استفاده از تابع get_bmap که خودمان تعریف کردیم و توضیح داده شد آدرس سکتور هر بلاک را برگردانیم، برای این کار اشاره گر به استراکت inode که با استراکت فایل که قبلا در تابع به دست آوردیم را به عنوان آرگومان اول و شماره بلاک را به عنوان آرگومان دوم می‌دهیم و این کار را تا زمانی که شماره بلاک کمتر از تعداد سایز بلاک های فایل است ادامه می‌دهیم و مقدار را در آرایه sectors ذخیره می‌کنیم. سایز بلاک های فایل را میتوان از طریق pf->ip->size/BSIZE که pf اشاره گر به استراکت فایل و ip اشاره گر به استراکت inode است، به دست آورد. تابع ()argptrint که گفته شد در واقع همان تابع ()argptr است با این تفاوت که پارامتر آرایه را به صورت int** میخواند نه char** و آن را در فایل syscall.c اضافه کردیم. توجه شود که امضای این تابع را به فایل defs.h اضافه کردیم.

پس از اجرای برنامه تست آن میبینیم که آدرس سکتور ها و بلاک ها به طور متوالی پشت هم قرار میگیرند و پس از اینکه آدرس سکتور هر فایل را می خواهیم چاپ کنیم می بینیم که توالی آن ۲ تا بیشتر می شود.

۷) توضیح دهید عدم توالی دادههای یک فایل روی دیسک، چگونه ممکن است در فرایند بازیابی آن پس از حذف مشکل ساز شود.

اگر داده های یک فایل روی دیسک متوالی نباشد ممکن است در فرایند بازیابی نتوانیم به داده ها دسترسی داشته باشیم و داده ها از دست بروند و همچنین اگر حجم زیادی از حافظه پر باشد این مشکل بیشتر بروز پیدا می‌کند و همچنین این کار از نظر زمانی و کارایی به صرفه نیست و نیاز به مدیریت حافظه زیادی خواهد داشت. توجه شود که اگر بلاک ها در آدرس ها متوالی نباشند دیگر نمی توان با استفاده از آدرس اول آن بلاک و سایز ها در ساختارهای مربوطه به بقیه اطلاعات دسترسی داشت و بنابراین باید آدرس های مختلفی را نگه داریم و تغییر در استراکت های سیستم عامل ایجاد خواهد شد. توجه شود که اگر آدرس سر یکی از بلاک ها در این حافظه های حذف شده باشد مشکلات زیادی برای بازیابی آن وجود خواهد داشت و شاید دیگر ممکن نباشد.

۳- پیاده سازی فراخوانی سیستمی get_parent_pid():

ابتدا مراحل اولیه ذکر شده در بالا را برای اضافه کردن فراخوانی سیستمی انجام می دهیم.

سپس در فایل سطح کاربر تابع get_parent_pid() را صدا میکنیم و مقدار بازگشتی یعنی pid پردازه پدر را چاپ میکنیم.

در نهایت تابع فراخوانی سیستمی مورد نظر را در فایل sysproc.c اضافه می کنیم. در این تابع به سادگی با استفاده از تکه کد myproc()->parent->pid شناسه پردازه پدر را بر میگردانیم.

۴- پیاده سازی فراخوانی سیستمی set_process_parent(int pid):

ابتدا مراحل اولیه ذکر شده در بالا را برای اضافه کردن فراخوانی سیستمی انجام می دهیم.

برای تست، دو فایل جدید با نام های test_A.c و test_D.c اضافه می کنیم که در test_A.c ابتدا با استفاده از سیستم کال fork، یک پردازه جدید تولید می کنیم و سپس pid خود این پردازه و pid پدر این پردازه را چاپ می کنیم. سپس با استفاده از سیستم کال sleep، به مدت ۲۰ ثانیه در اجرای این دستور وقفه ایجاد می کنیم و سپس دوباره pid خود این پردازه و پدر این پردازه را چاپ می کنیم.

در test_D.c، ابتدا توسط argv[1] مقدار pid پردازه ای که در test_A.c ساخته ایم را به عنوان ورودی می دهیم و سپس با استفاده از سیستم کال set_process_parent، این پردازه را به عنوان پدر پردازه فعلی می گذاریم. نمونه ای از اجرای برنامه به شکل زیر است:

```
$ test_A &
$ in sys_get_parent_pid

pid: 5, parent: 1
test_D 5
in sys_set_process_parent

in set_process_parent --> input pid : 5

in set_process_parent --> myproc.pid : 6, myproc.process_parent : 5
$ in sys_get_parent_pid

pid: 5, parent: 1
```

حال تابع فراخوانی سیستمی مورد نظر را در فایل sysproc.c اضافه می کنیم. در این تابع به کمک argint پارامتر ورودی از روی stack برداشته می شود و به تابع کمکی آن به نام set_process_parent که در فایل proc.c تعریف شده است، پاس داده می شود و مقدار بازگشتی این تابع که 0 یا -1 می باشد به عنوان خروجی تابع ریترن می شود، در واقع در صورتی که که این تابع pid مورد نظر را پیدا کند و عملیات موفقیت آمیز باشد مقدار 0 و در غیر این صورت مقدار -1 را برمیگردانیم.

در تابع set_process_parent در فایل proc.c مشابه فراخوانی سیستمی kill ابتدا از دستور aquire(&ptable.lock) استفاده می شود تا به لیست پردازه های در حال اجرا دسترسی داشته باشیم و سپس روی آرایه ptable پیمایش می کنیم و در صورتی که پردازه ای با pid مورد نظر پیدا کنیم، آن پردازه را به process_parent پردازه کنونی اساین میکنیم و به کمک دستور release(&ptable.lock) آرایه

پردازه ها را آزاد می کنیم و مقدار 0 را ریترن میکنیم و در صورتی که چنین پردازنده ای پیدا نشود ابتدا به کمک دستور release(&ptable.lock) آرایه پردازه ها را آزاد میکنیم و در نهایت مقدار -1 را ریترن می کنیم.(پروتوتایپ این تابع را در فایل defs.h اضافه می کنیم)

شناسه آخرین کامیت:

ba9137b2e6c67455cade6d210914f41afecd9dbe

آدرس مخزن:

https://gitlab.com/ali_akhgari/os-lab