**به نام خدا**

**گزارش پروژه «3» آزمایشگاه سیستم عامل**

**استاد: دکتر کارگهی**

**گروه 11**

**امیرارسلان شهبازی 810101451**

**سید محمدحسین مظهری 810101520**

**محمدمهدی صمدی 810101465**

**لینک مخزن** [**https://github.com/AMIRSH1383/OS-SMS\_LAB3.git**](https://github.com/AMIRSH1383/OS-SMS_LAB3.git)

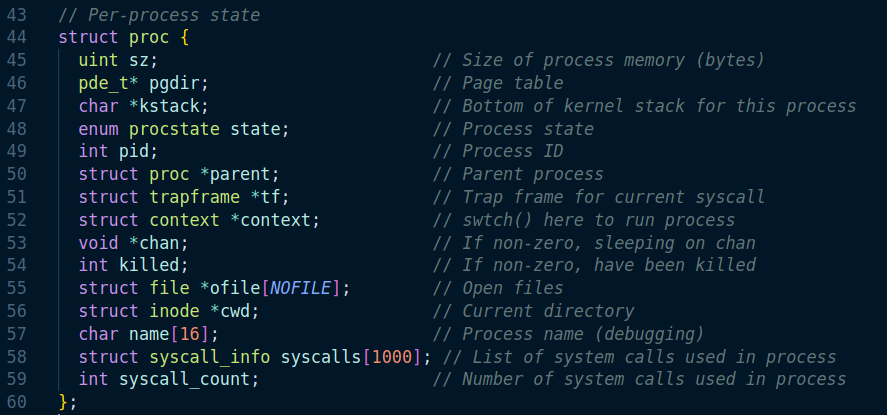
**آخرین کامیت 1d937ba33775d174c0fd2d51879395c4da638e44**

**1) ساختار PCB و همچنین وضعیت های تعریف شده برای هر پردازه را در xv6 پیدا کرده و گزارش کنید. آیا شباهتی میان داده های موجود در این ساختار و ساختار به تصویر کشیده شده در شکل 3.3 منبع درس وجود دارد؟**

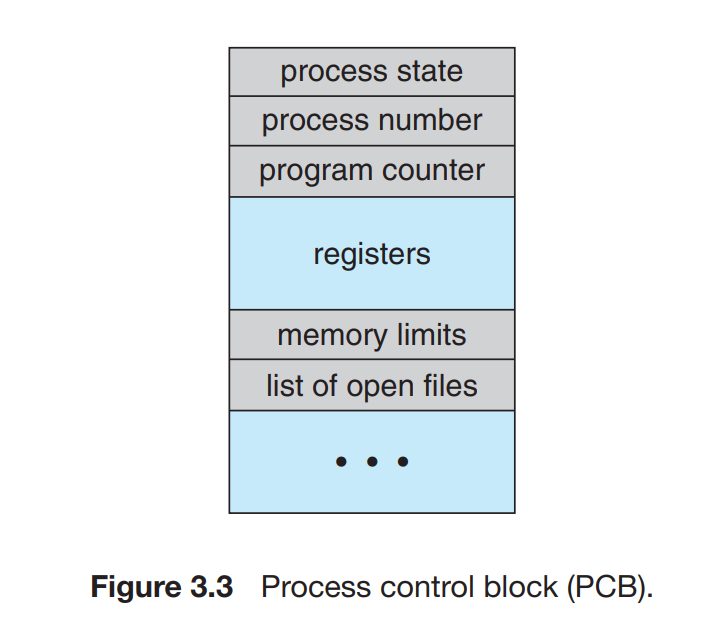
در فایل proc.h استراکت هایی که برای یک پراسس به کار می روند تعریف شده است.

همانطور که در شکل زیر مشاهده می کنید، داده ساختار زیر برای پراسس های ما وجود دارند.

توضیحات مربوط به هر مشخصه به صورت کامنت جلوی آن داده شده است و ما صرفا شباهت های آن نسبت به شکل منبع درس را گزارش خواهیم کرد.



شکل منبع درس به صورت زیر است.



همانطور که می بینید شباهت هایی میان این دو ساختار وجود دارد.

در بخش زیر معادل هریک از ویژگی های جدول بالا در xv6 را می آوریم.

process state => enum procstate state

process number => int pid

registers => struct context \*context;

در استراکتی که برای کانتکست تعریف می کنیم از اسامی رجیسترها استفاده و می کنیم به همین دلیل برای رجیسترها کانتکست را مشابه در نظر می گیریم.

Memory limits => uint sz

List of open file => struct file \*ofile [*NOFILE*]

Memory management information => pde\_t\* pgdir

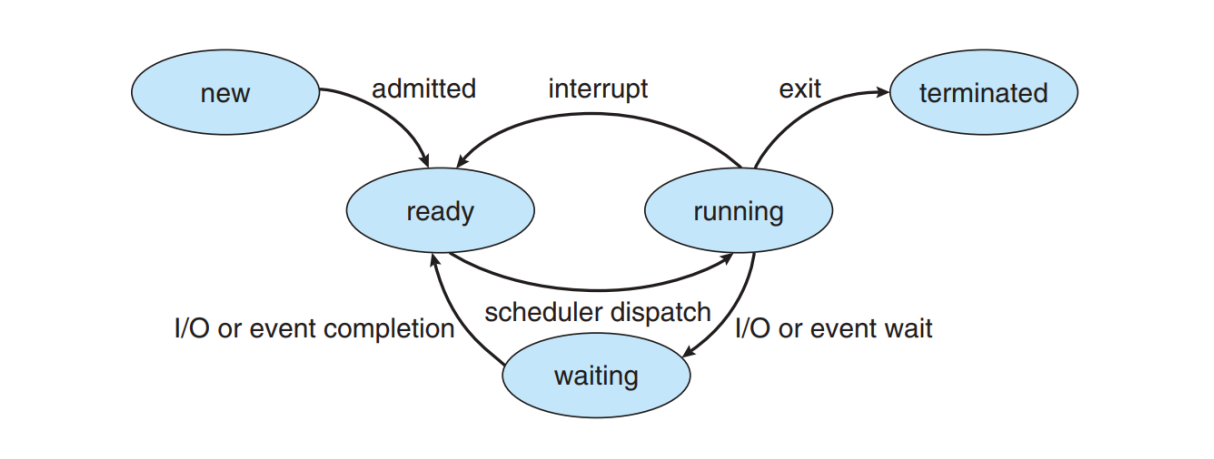
اما بخش دیگری که به آن باید بپردازیم وضعیت هایی است که هر استیت می تواند در آن قرار بگیرد.

این وضعیت ها مجدد در همان فایل چند خط بالاتر تعریف شده که به شرح زیر است.



در سوال بعد توضیحات مربوط به هریک را می دهیم.

**2) هرکدام از وضعیت های تعریف شده معادل کدام وضعیت در شکل 1 می باشد؟**



new => EMBRYO

ready => RUNNABLE

running => RUNNING

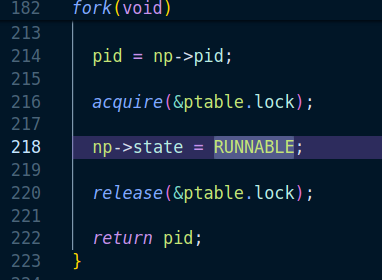
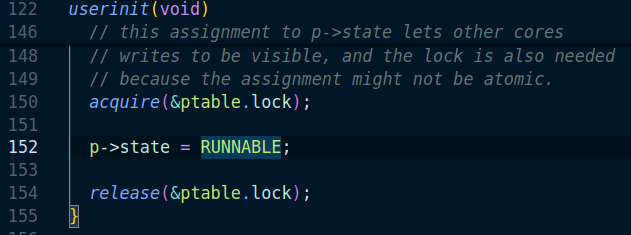
waiting => SLEEPING

terminated => ZOMBIE

یک استیت باقی ماند که در بالا تعریف نشده بود که آن استیت UNUSED است توضیحات این استیت به این صورت است که این وضعیت نشان‌دهنده این است که ورودی برای اختصاص به یک فرآیند جدید در دسترس است اما در حال حاضر در حال استفاده نیست.(به مرحله new نرسیده است)

**3) با توجه به توضیحات گفته شده، کدام یک از توابع موجود در proc.c منجر به انجام گذار از حالت new به حالت ready که در شکل 1 به تصویر کشیده شده، خواهد شد؟ وضعیت یک پردازه در xv6 در این گذار از چه حالتی به چه حالتی تغییر می کند؟ پاسخ خود را با پاسخ سوال 2 مقایسه کنید.**

همانطور که در شکل زیر مشاهده می کنید ما دوجا این کار را کردیم. اولی برای ساخت اولین پراسس کاربر است. و دومی که در بقیه موارد استفاده می شود در تابع fork است که برای ساخت پردازه جدید به کار می رود. کد به این صورت است که در پایان وضعیت پراسس در حال ساخت را به پراسس در حال اجرا تبدیل می کند.

پس با توجه به این موارد جمع بندی ما این می شود که وضعیت ما از ابتدای کار که در EMBRYO یا همان نیو قرار داشتیم به وضعیت RUNNABLE یا همان ready تبدیل می شود که این قضیه با مواردی که از سوالات یک و دو دریافته بودیم همخوانی دارد.

**4) سقف تعداد پردازه های ممکن در xv6 چه عددی است؟ در صورتی که یک پردازه تعداد زیادی پردازه فرزند ایچاد کند و از این سقف عبور کند کرنل چه واکنشی نشان داده و برنامه سطح کاربر چه بازخوردی دریافت می کند؟**

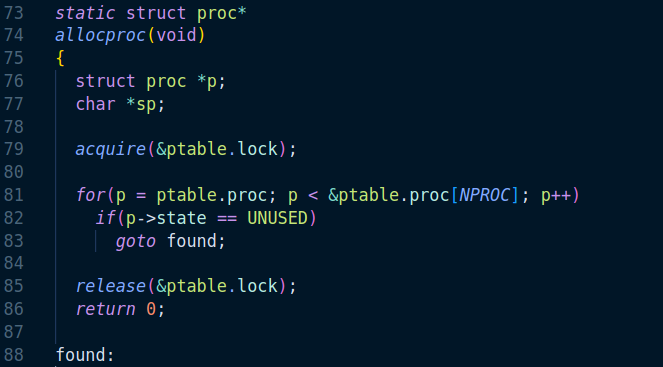
سقف تعداد پردازه ها به صورت یک مقدار دیفاین شده در فایل param.h قرار گرفته است.



همانطور که مشاهده می کنید حداکثر 64 پراسس می تواند در لحظه وجود داشته باشد.

با توجه به کدی که برای اختصاص دادن پراسس نوشته شده است کرنل نسبت به ساخت بیش از حد پراسس ها واکنش نشان می دهد. این عملیات به این صورت است که اگر ptable ما دیگر ظرفیت برای اختصاص دادن پراسس جدید را نداشت قفل آزاد شده و ریترن 0 خواهیم داشت که این ریترن نشان دهنده این است که در اختصاص پراسس خطا رخ داده و نتوانستیم پراسس جدید ایجاد کنیم.

اما اگر جا برای این کار داشتیم همانطور که مشاهده می کنید از عملیات goto استفاده کردیم و ریترن 0 را رد کردیم و پراسس را در جلوتر تشکیل داده و پوینتری به پراسس ساخته شده را برمی گردانیم.



**5) چرا نیاز است در ابتدای هر حلقه تابع schedule، جدول پردازه ها قفل شود؟ آیا در سیستم های تک پردازه ای هم نیاز است این کار صورت بگیرد؟**

این به این دلیل است که مطمئن باشیم در هرلحظه تنها یک پردازه یا ترد به جدول پردازه ها دسترسی دارد. اگر اینطور نبود یک رقابتی میان پردازه ها به وجود می آمد که اصطلاحا به آن race condition می گویند به این صورت که ممکن بود در لحظه چند پراسس به جدول پردازه ها دسترسی می داشتند که این منجر می شد که داده های ما خراب و ناپایدار باشند.

اگر سیستم تک پردازه ای بود از اینکه در هر لحظه در سیستم تنها یک پردازه وجود دارد مطمئن بودیم و شرایط رقابتی به وجود نمی آمد لذا نیازی نبود جدول پردازه ها را قفل کنیم.

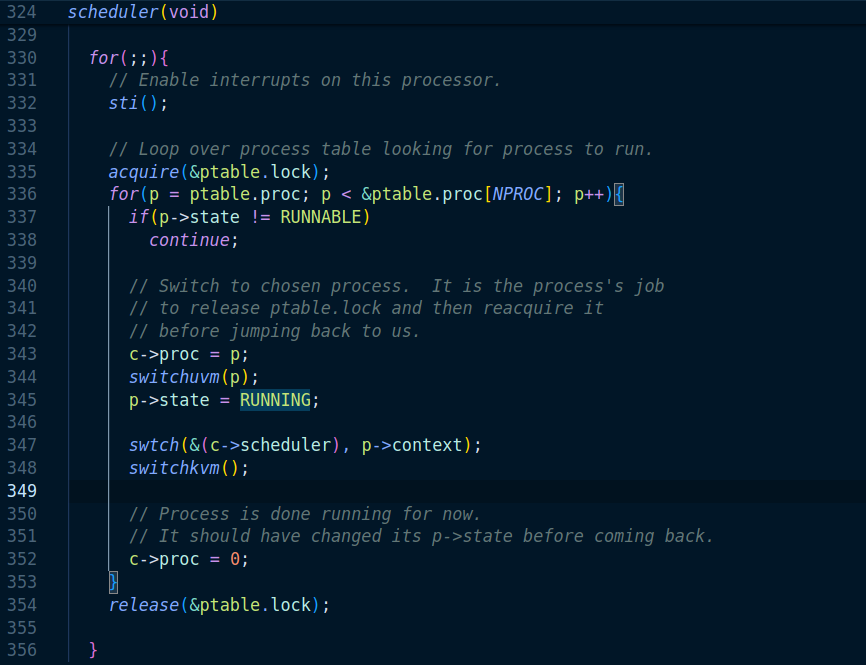
**6) با فرض اینکه xv6 در حالت تک هسته ای در حال اجراست، اگر یک پردازه به حالت runnable برود و صف پردازه ها در حال طی شدن باشد، در مکانیزم زمان بندی xv6 نسبت به موقعیت پردازه در صف، در چه iteration ای امکان schedule پیدا می کند؟**

در تابع scheduler طبق قطعه کد زیر مشاهده می کنیم که وقتی که روی صف پردازه ها پیمایش می کنیم دنبال این می گردیم که یک پراسسی که در وضعیت runnable باشد را پیدا کنیم.

همانطور که می بینید کد ما به این صورت است که اگر پراسسی قابل اجرا نباشد، از آن عبور می کند تا به یک پراسس با قابلیت اجرا برسد.

هر وقت که این پراسس را پیدا کرد، سپس context switch انجام داده تا پراسس runnable به پراسس running تبدیل بشود.

پس از همه ی این مراحل وقتی کارمان تمام می شود قفل جدول پردازه ها را رها می کنیم و مجدد این عملیات ها انجام می شوند.

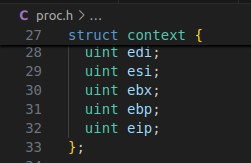


**7)**

توضیحاتی درباره struct context:

* برای ذخیره حالت registerها در لحظه context switch استفاده می‌شود.
* تمام register ها ذخیره نمی‌شوند چرا که برخی از آن‌ها مقدار ثابتی در تمام موضوعات دارند.
* نیازی به ذخیره registerهایی مثل EAX, ECX, EDX نداریم زیرا قرارداد طراحی XV6 تضمین می‌کند که که توسط صداکننده تابع ذخیره شده‌اند.
* برای هر استک، ساختار کانتکستش در انتهای پایین استک قرار دارد. پس ESP به مکان این ساختار اشاره می‌کند.

توضیحات هر register:

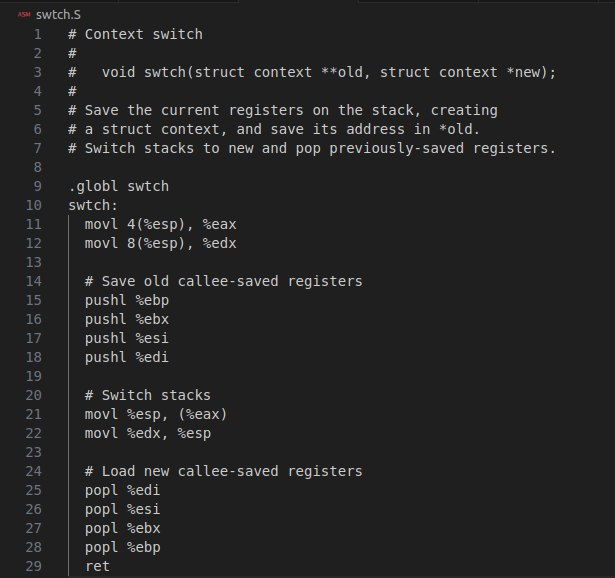


* EDI: مخفف extended destination index است. این رجیستر عمدتا برای عملیات‌های مربوط به string استفاده می‌شود. معمولا pointer به مقصد در هنگام عملیات‌هایی مثل MOVS, CMPS, SCAS و غیره را نگه می‌دارد.
* ESI: مخفف extended source index است. این register نیز برای عملیات‌های مربوط به string استفاده می‌شود. اما تفاوتش با EDI این است که pointer به مبدا را نگه می‌دارد. یعنی آدرس جایی که string از آن خوانده می‌شود را دارد.
* EBX: مخفف extended base register است. عموما برای نگه‌داری داده، شمارنده و آدرس‌ها استفاده می‌شود.
* EBP: مخفف extended base pointer است. برای نگه‌داری pointer به base استک کنونی استفاده می‌شود. کمک می‌کند که طبق قراردادهایی که داریم، به پارامترها و متغیرهای محلی تابع دسترسی پیدا کنیم. طبق قرارداد، این register در تمام مدت زمان اجرای یک تابع ثابت باقی می‌ماند و راحت‌تر می‌توانیم بدانیم هر متغیر کجا قرار دارد.
* EIP: مخفف extended instruction pointer است. به instruction بعدی که قرار است اجرا شود اشاره ‌می‌کند. در هر بار اجرای instruction جدید این عدد افزایش میابد.

**8)**

در struct context پردازنده XV6، مقدار program counter در EIP ذخیره می‌شود. در واقع این register آدرس دستور بعدی که باید اجرا شود را نگه می‌دارد.

چگونگی انجام این کار را باید در کد اسمبلی نوشته شده بررسی کنیم. کد به صورت زیر است:



* خط 9 تا 12: تابع دو آرگومان می‌گیرد. طبق قرارداد می‌دانیم آرگومان ها بلافاصله بعد از اشاره‌گر ESP قرار دارند. پس آدرس قدیمی در ESP+4 قرار دارد که به رجیستر EAX می‌رود. آدرس جدید هم که در ESP+8قرار دارد به رجیستر EDX می‌رود.
* خط 15 تا 18: مقدار کنونی رجیسترهای struct context (به جز EIP که برای PC است) به استک وارد می‌شوند.
* خط 21 و 22: آدرس استک کنونی که در ESP است، در آدرسی که رجیستر EAX اشاره می‌کند ذخیره می‌شود. در واقع این آدرس همان آدرس context قدیمی است که در آرگومان تابع داده شده بود. سپس محتوای جدید که در EDX ذخیره‌اش کرده بودیم، به ESP که استک کنونی را نشان می‌دهد انتقال میابد.
* خط 25 تا 29: حالا مقدار رجیسترهای struct context کنونی برعکس ترتیب پوش شدن‌شان از استک پاپ می‌شوند و در رجیسترهای EDI, ESI, EBX و EBP ذخیره می‌شوند. در آخر ret از استک return address را پاپ می‌کند و در EIP می‌گذارد. بدین ترتیب PC به درستی آپدیت می‌شود.

**9)**

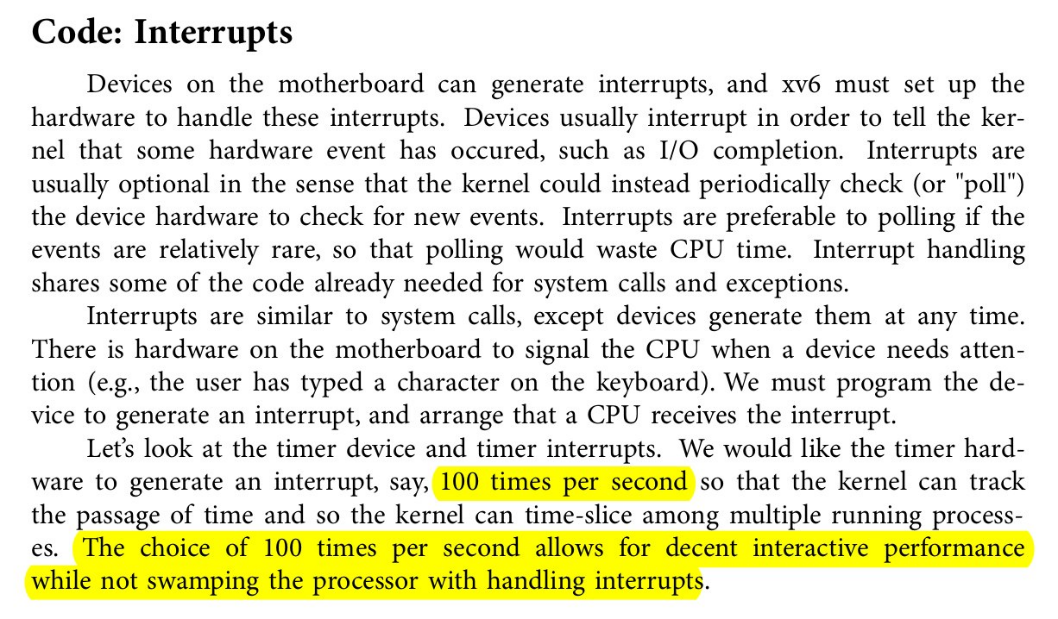
عواقب فعال نکردن interrupt ها:

* پاسخ ندادن به hardware interruptها: در این صورت کارهای مهمی مثل ورودی کیبورد، ماوس یا تایمر نادیده گرفته می‌شوند.
* نداشتن time sharing: در این حالت امکان اجرای preemptive را نداریم. پس هر process از زمانی که CPU را بگیرد تا زمانی که کارش تمام نشود، CPU را پس نمی‌دهد.

**10)**

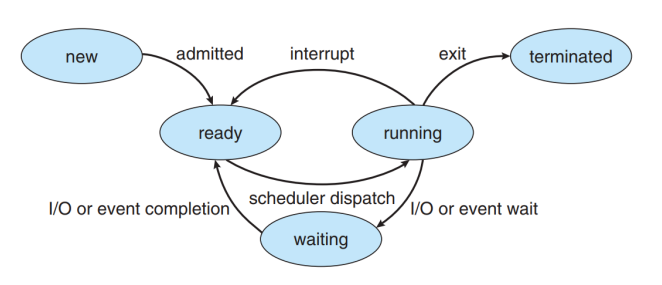
فاصله زمانی تو timer interrupt متوالی:

برای این سوال به توضیحات کتاب XV6 مراجعه می‌کنیم. طبق متن در هر ثانیه 100 بار فعال می‌شود. پس فاصله زمانی دو وقفه متوالی 10 میلی ثانیه است.



**11)**

تابعی که منجر به تغییر حالت به علت وقفه می‌شود:

به شکل 1 صورت پروژه مراجعه می‌کنیم. این تابع باید ما را از حالت running به حالت runnable ببرد. طبق توضیحات صورت پروژه تابع yield این وظیفه را به عهده دارد****

**12)**

مقدار کوانتوم زمانی الگوریتم Round Robin:

طبق توضیحات سوال 10، هر 10 میلی ثانیه وقفه زمان فعال می‌شود. از آن جایی که با هر بار فعال شدن وقفه زمانی، کوانتوم زمانی یک process به پایان می‌رسد می‌توان نتیجه گرفت که مقدار کوانتوم زمانی برابر 10 میلی ثانیه است. اما در پیاده‌سازی که در سطح اول صف انجام می‌دهیم این مقدار برابر 50 میلی ثانیه خواهد شد چرا که با هر 5 بار فعال شدن وقفه زمانی کوانتوم گرفته می‌شود.

**13) تابع wait در نهایت از چه تابعی برای منتظر ماندن برای اتمام کار یک پردازه استفاده می کند ؟**

* این تابع waitاست. این تابع به پردازه والد امکان می‌دهد برای اتمام پردازه‌های فرزند خود منتظر بماند.
* عملکرد تابع به این صورت است که والد منتظر می‌ماند تا یکی از فرزندانش به حالت ZOMBIEبرسد. پس از آن، اطلاعات مربوط به پردازه فرزند (مانند PID و کد خروج) به والد بازگردانده شده و منابع فرزند از سیستم حذف می‌شوند.
* اگر فرزندی وجود نداشته باشد یا همه فرزندان قبلاً جمع‌آوری شده باشند، تابع wait مقدار -1 بازمی‌گرداند.

**14) با توجه به پاسخ سوال قبل ، استفاده(های) دیگر این تابع چیست ؟ (ذکر یک نمونه)**

تابع wait علاوه بر آزادسازی منابع پردازه‌های فرزند که به پایان رسیده‌اند، یک کاربرد مهم دیگر نیز دارد:

جلوگیری از پردازه‌های زامبی

* زمانی که یک پردازه فرزند به پایان می‌رسد، وضعیت آن به ZOMBIE تغییر می‌کند.
* پردازه در حالتZOMBIE باقی می‌ماند تا زمانی که پردازه والد اطلاعات وضعیت نهایی آن (مثل exit code ) را دریافت کند.
* اگر پردازه والد از wait استفاده نکند، پردازه فرزند به حالت زامبی باقی می‌ماند و منابع آن (مانند جدول پردازه‌ها) آزاد نمی‌شوند.

چگونه wait از پردازه‌های زامبی جلوگیری می‌کند؟

1. وقتی والد تابع wait را فراخوانی می‌کند:
   * وضعیت پردازه فرزند را دریافت می‌کند.
   * منابع آن پردازه در کرنل آزاد می‌شوند.
2. این کار تضمین می‌کند که هیچ پردازه زامبی‌ای در سیستم باقی نماند.

**15) با این تفاسیر ، چه تابعی در سطح کرنل ، منجر به آگاه سازی پردازه از رویدادی است که برای آن منتظر بوده است ؟**

* در xv6، تابعی که برای آگاه‌سازی پردازه از وقوع رویدادها استفاده می‌شود، wakeup است.
* وقتی پردازه‌ای منتظر یک رویداد (مانند اتمام I/O) است، در حالت SLEEPING قرار می‌گیرد. زمانی که رویداد مورد نظر رخ دهد، wakeup همه پردازه‌هایی را که در انتظار آن رویداد خاص بوده‌اند، به حالت RUNNABLE منتقل می‌کند.

**16) با توجه به پاسخ سوال 9 ، این تابع منجر به گذار از چه وضعیتی به چه وضعیتی در شکل 1 خواهد شد ؟**

* تابع wakeup باعث تغییر وضعیت پردازه از waiting به ready می‌شود.
* به این ترتیب: پردازه‌ای که در حالت waiting بوده (منتظر رویدادی مانند اتمام خواندن فایل)، پس از وقوع رویداد مورد نظر به حالت ready تغییر وضعیت داده و آماده اجرا می‌شود.

**17) آیا تابع دیگری وجود دارد که منجر به انجام این گذار شود؟ نام ببرید .**

علاوه بر wakeup، تابع wake نیز وجود دارد که در xv6 برای بیدار کردن یک پردازه خاص استفاده می‌شود.

تفاوت بین این دو تابع:

* wakeup همه پردازه‌هایی که منتظر یک رویداد خاص هستند را بیدار می‌کند.
* wake یک پردازه خاص (براساس ساختار داده struct proc) را از حالت SLEEPING به RUNNABLE تغییر می‌دهد.

**18) در بخش 3.3.2 منبع درس با پردازه های Orphan آشنا شدید ، رویکرد xv6 در رابطه با این گونه پردازه ها چیست ؟**

تعریف پردازه orphan ( یتیم):

یک پردازه یتیم (Orphan Process) پردازه‌ای است که والد آن (Parent) پیش از اتمام اجرای پردازه فرزند، خاتمه یافته باشد. در این شرایط، پردازه یتیم بدون والد باقی می‌ماند و نیازمند مدیریتی خاص توسط سیستم‌عامل است.

رویکرد xv6 نسبت به پردازه‌های Orphan :

در xv6، وقتی یک پردازه والد خاتمه پیدا می‌کند:

1. تمام پردازه‌های فرزند آن (اعم از پردازه‌های فعال، معلق، یا زامبی) به پردازه initproc منتقل می‌شوند.
2. Initproc به عنوان والد جدید این پردازه‌های یتیم عمل می‌کند و مسئولیت مدیریت آن‌ها را بر عهده می‌گیرد.

جزئیات عملکرد:

* زمانی که والد پردازه‌ای حذف می‌شود، کرنل xv6 حلقه‌ای در جدول پردازه‌ها اجرا می‌کند. این حلقه هر پردازه‌ای که به پردازه خاتمه‌یافته به‌عنوان والد وابسته باشد، پیدا کرده و والد آن را به initproc تغییر می‌دهد.
* اگر پردازه یتیم به حالت ZOMBIE برسد، initproc وظیفه جمع‌آوری و آزادسازی منابع آن را دارد.

چرا initproc؟

* Initproc اولین پردازه‌ای است که در سیستم توسط کرنل اجرا می‌شود و همیشه فعال باقی می‌ماند.
* به همین دلیل، از این پردازه به‌عنوان والد جایگزین برای مدیریت پردازه‌های یتیم استفاده می‌شود.

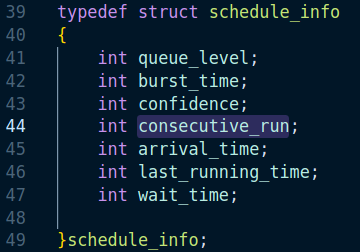
**زمان بندی بازخوردی چندسطحی**

در ابتدا ما می بایست یکسری ساختار جدید به ساختار پراسس اضافه کنیم که مربوط به کارهای زمان بندی است.

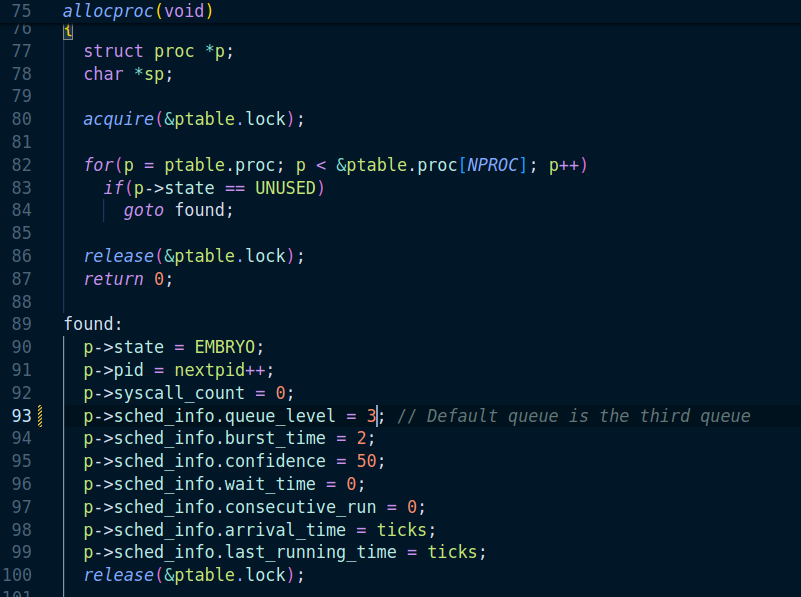
با توجه به اینکه سه سطح الگوریتم داریم لذا یک enum در فایل proc.h برای این سه الگوریتم داریم که به شکل زیر است.



سپس یک استراکت جدید به نام schedule\_info می سازیم که اطلاعاتی که در زمان بندی برای ما مهم هستند را در آن قرار می دهیم. این استراکت به شرح زیر است.



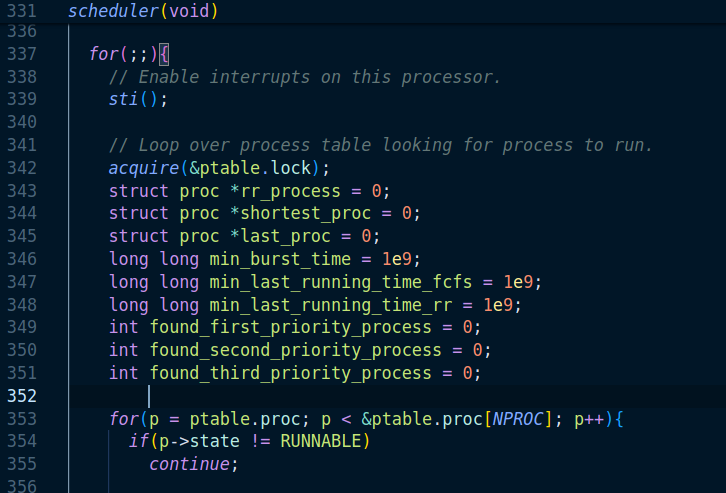
گام بعدی این است که نمونه ای از این استراکت را در استراکت پراسس قرار می دهیم. اینگونه هر پراسسی که ساخته می شود اطلاعاتی برای زمان بندی را نیز درون خود نگه می دارد. مقادیر اولیه ی این خصیصه ها به شرح زیر است.



این مقادیر اولیه هنگام تخصیص پراسس به خصیصه ها تعلق می گیرد.

بخش عمومی به اتمام رسید و از الان به پیاده سازی هر الگوریتم می پردازیم.

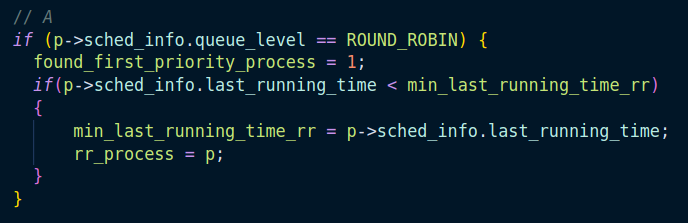
**سطح اول: زمان بند نوبت گردشی با کوانتوم زمانی**



در ابتدای حلقه بی نهایتی که در فایل زمان بند وجود دارد یکسری متغیرهایی که قرار است استفاده کنیم را تعریف می کنیم. سه تا پراسس تعریف شده که هر کدام برای خود پراسس مربوط به همان سطح را اخذ کند و بسته به وزن هرکدام از پراسس مربوطه استفاده کنیم.

یکسری متغیرهایی مانند برخی زمان ها که در الگوریتم های اول و سوم به کار می رود نیز تعریف شده اند.

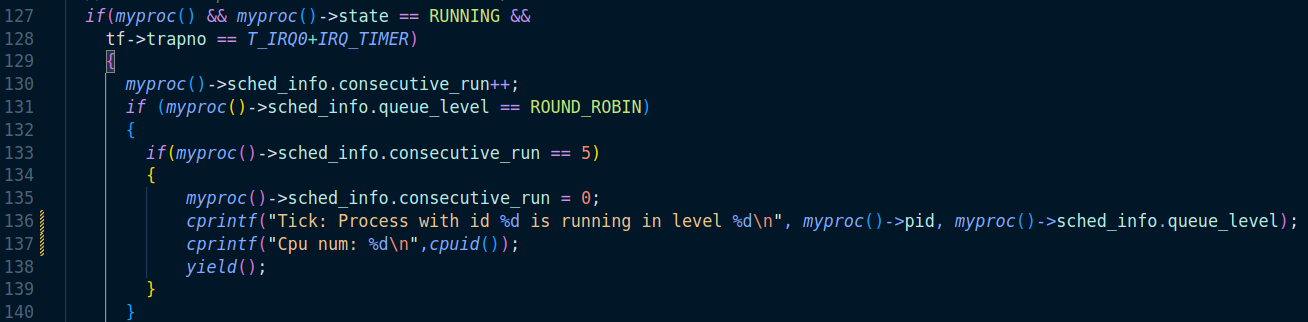
در نهایت هم برای هر پراسسی که از هر سطح پیدا می کنیم مشخص می کنیم اگر پراسسی در آن سطح وجود دارد یا نه.



در ادامه حلقه بالا ما می بایست برای هریک از الگوریتم هایی که داریم یک شرط بگذاریم و ببینیم که آیا پراسسی که داریم پیمایش می کنیم عضو این الگوریتم هست یا خیر.

در سطح اول اگر پراسس جزو سطح اول بود، می گوییم یک پراسس سطح اول پیدا کردیم و در ادامه چک می کنیم اگر بیشتر از بقیه از مدت زمان اجرا شدنش گذشته است این نشان می دهد که بیشتر از بقیه از زمان اجرا شدنش گذشته و الان وقت اجرای آن است. پس مینیمم اخرین زمان اجرا را مجدد مقدار دهی می کنیم و پراسس نوبت گردشی را ست می کنیم.

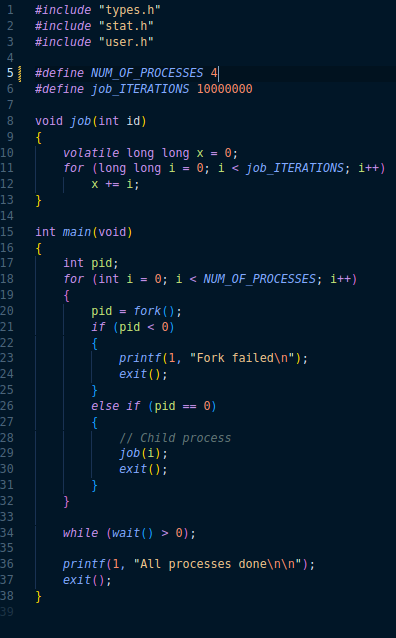
در نهایت از انجایی که هر تیک به اندازه 10 میلی ثانیه در نظر گرفته شده است و ما می خواهیم که کوانتوم زمانی ما 50 میلی ثانیه باشد، لذا می بایست هر 5 اینتراپت یک بار تابع yield صدا زده شود که این امر را در فایل trap.c اعمال می کنیم که به شکل زیر می شود.



همانطور که مشاهده می کنید ما یک چیزی تعریف کردیم به عنوان تعداد دفعات متوالی اجرا که به این صورت است که اگر در نوبت گردشی به پنج رسید یعنی انگار پنج تیک خورده است و ما در این زمان yield را صدا می زنیم.

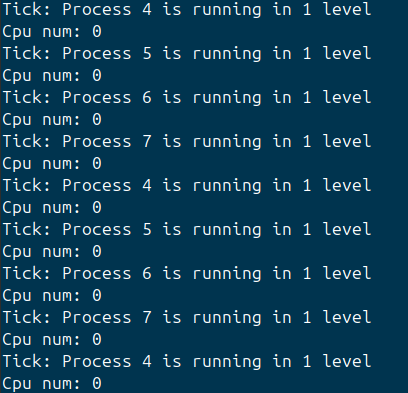
در نهایت هم به کمک تابع سطح کاربری که نوشتیم تست می کنیم که هر یک از پراسس ها به چه صورت اجرا می شوند.

تابع سطح کاربر را در فایل schedule\_test نوشتیم که به شکل زیر است. یکسری پراسس به تعدادی که مشخص می کنیم می سازد و یک حلقه طولانی اجرا می کند.(به جای sleep از حلقه طولانی استفاده کردیم)



در انتها هم این فایل را برای دو تعداد پردازنده یک و دو اجرا می کنیم و نتیجه را مشاهده می کنیم.

برای تک پردازنده ای داریم:

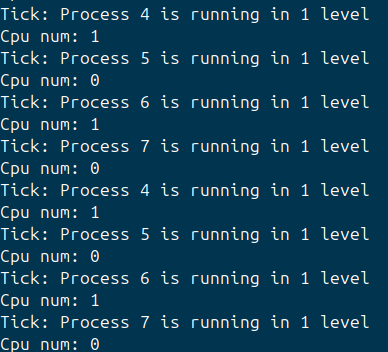


همانطور که می بینید الگوی تکراری نوبت گردشی دارد رعایت می شود و همه پراسس ها روی پردازنده صفرم اجرا می شوند.

**19)**

اما اگر تعداد پردازنده ها را به دو برگردانیم دو اتفاق می افتد. اول اینکه پراسس ها بین دو پردازنده تقسیم می شوند.(مانند شکل پایین) و همچنین سرعت پردازش بیشتر شد و کار ما زودتر برای همان پراسس ها به اتمام رسید.

لذا اگر هرپراسس را بخواهیم بررسی کنیم انگار 4و 6 و بعد 5 و 7 اجرا می شوند. اما جایی که ما پرینت می کنیم به ازای هر تیک کلاک است که مشخص می کند هر پردازنده در چه وضعیتی است.



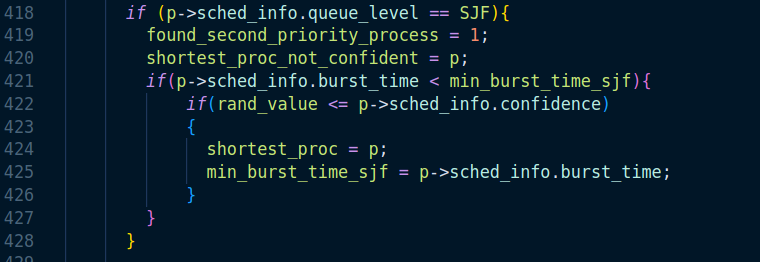
پس مشخص کردیم که سطح اول که نوبت گردشی بود به درستی کار می کند.

نکته پایانی: تست دیگری هم برای دو پردازنده انجام دادیم به این صورت که این بار فرد تا پراسس تولید کردیم مثلا 5 تا. وقتی اجرا کردیم مشاهده کردیم که یک ترتیب اجرا بین پردازنده ها می بینیم. یعنی فرض کنید سه پردازه و دو پردازنده داشته باشیم ترتیب پراسس و پردازنده به شرح زیر بود.

(1,1) (2,2) (3,1) (1,2) (2,1) (3,2), …

همانطور که می بینید در هر دور هر پراسس یکبار روی پردازنده یک و بار دیگر روی پردازنده دو اجرا می شود.

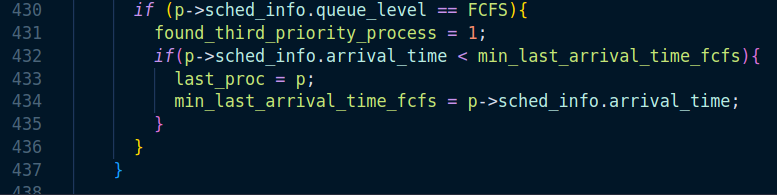
**سطح دوم: اول، کوتاه ترین کار**

****

در ادامه ایف مرحله قبل، این دفعه با توجه به الگوریتم کوتاه ترین کار چک می کنیم که آیا مقدار تصادفی ای که ساختیم از اطمینان ما کمتر است یا خیر. اگر بود آن پراسس را انتخاب می کنیم در غیر این صورت سراغ پراسس بعدی می رویم.

مقدار تصادفی هم برگرفته از یک تابع دستی است که ما نوشتیم. این تابع براساس تیکی که در آن لحظه در آن قرار داریم کار می کند لذا تصادفی است.

**سطح سوم: اولین ورود-اولین رسیدگی**



در این بخش ما یک خصیصه ای داریم به نام آخرین زمانی که یک پراسس به صف ready رسیده است. پس به ترتیبی که پراسس ها رسیده اند پراسس انتخاب شده و اجرا می شود که این همان الگوریتم اولین ورود اولین رسیدگی است.

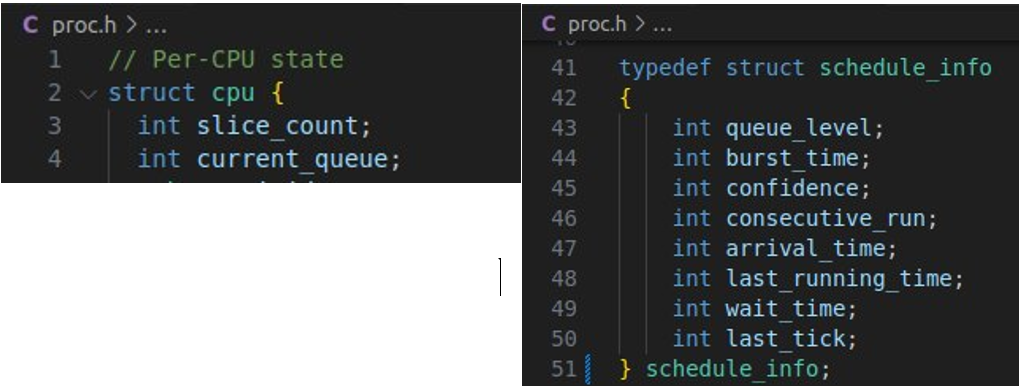
**برش دهی زمانی و ساز و کار افزایش سن**

در ابتدا struct schedule info را تعرییف می‌کنیم و به عنوان فیلد جدید struct proc قرار می‌دهیم.

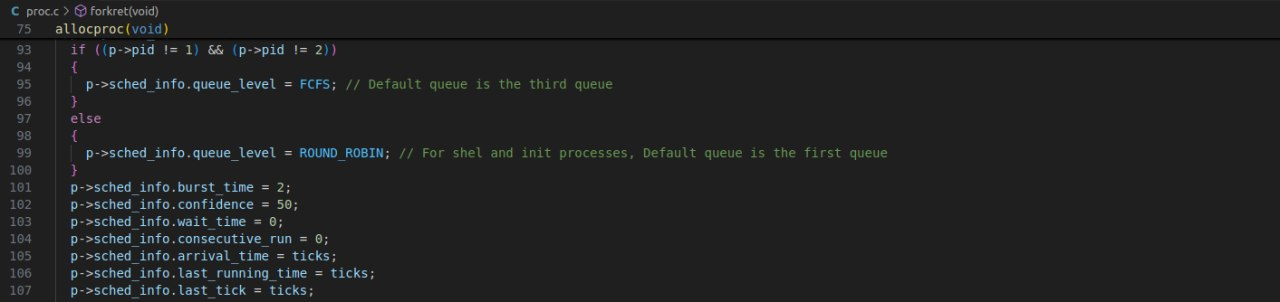
* Queue level: صفی که در آن قرار دارد.
* Burst time: زمان اجرا که به طور پیش فرض برای همه 2 در نظر گرفته شده.
* Confidence: اطمینان از اجرا که به طور پیش فرض 50 است.
* Consecutive run: تعداد کوانتوم‌های اجراهای متوالی.
* Arrival time: کلاک در زمان اولین بار رسیدن این پردازه به ready queue.
* Last running time: کلاک در زمان آخرین بار رسیدن این پردازه به ready queue.
* Wait time: تعداد کلاک‌هایی که این پردازه در ready queue مانده.
* Last tick: آخرین کلاکی که wait time این پردازه آپدیت شده است.

همچنین نیاز به تغییراتی در struct cpu داریم که در تصویر می‌بینیم.

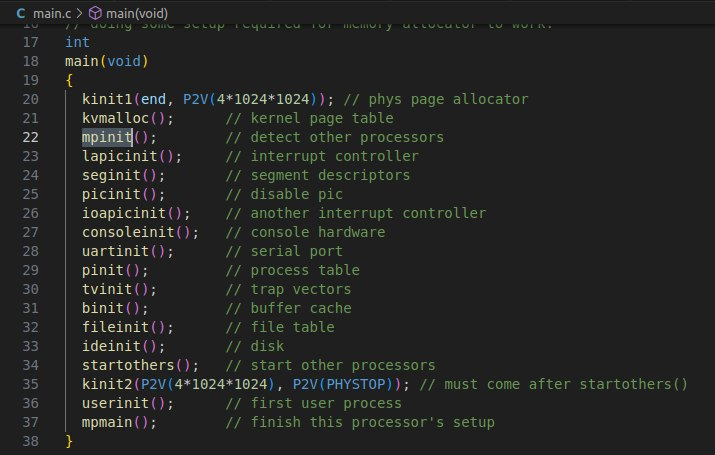
* Slice count: تعداد کلاک‌هایی که در صف فعالی ماندیم.
* Current queue: صف فعلی.



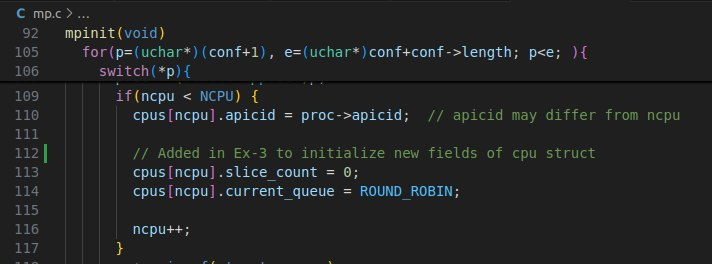
مقداردهی اولیه به فیلدهای جدید پردازه در تابع allocproc فایل proc.c انجام شده است.



برای مقداردهی فیلدهای جدید cpu به فایل main.c مراجعه می‌کنیم. از اینجا متوجه می‌شویم برخی فیلدهای cpu در تابع mpinit مقداردهی شده است.



پس ما هم فیلدهای جدید را آنجا می‌بریم.

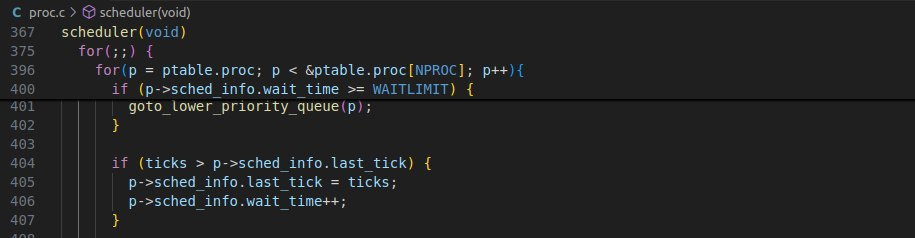


تغییرات تابع scheduler فایل proc.c:

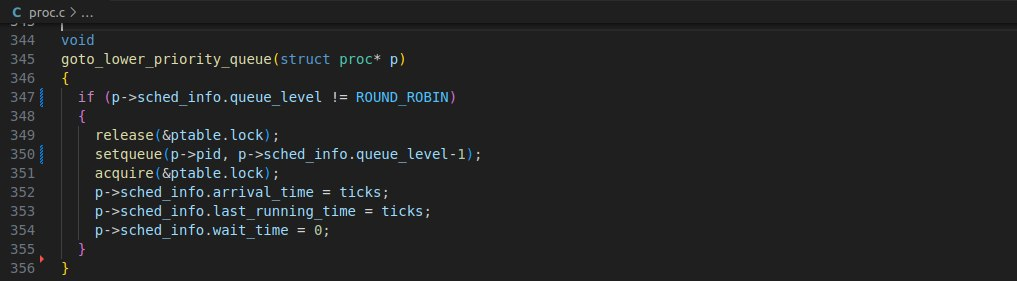
در ابتدای حلقه بی‌نهایت، متغیرهایی برای انتخاب پردازه مطابق سیاست خواسته شده اضافه شده‌اند.



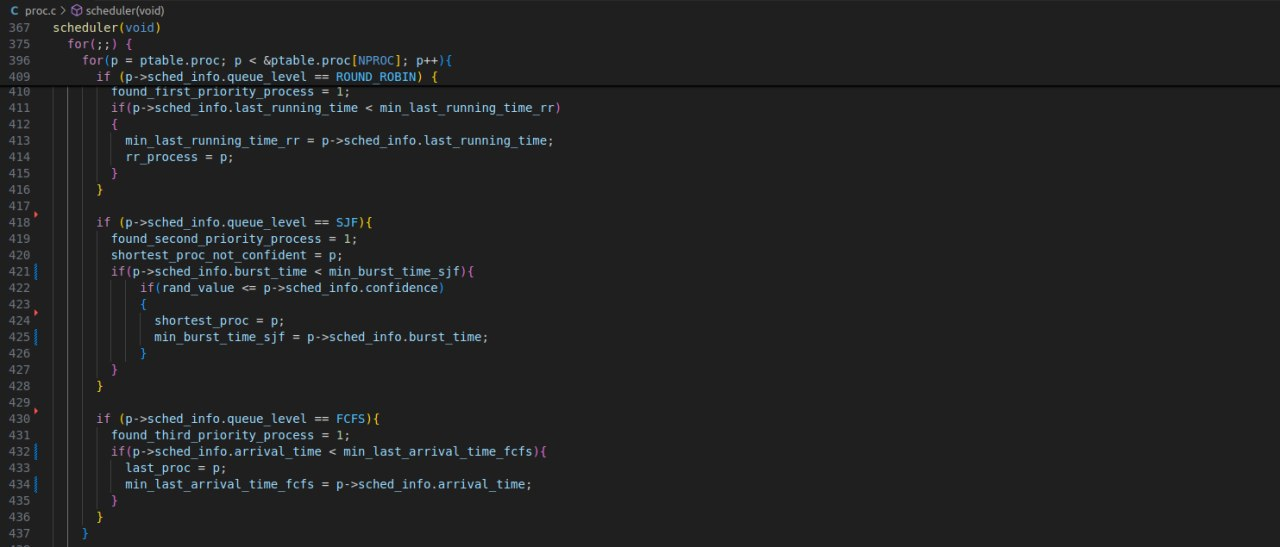
سپس روی تمام پردازه‌ها حلقه زده شده. در ابتدای این حلقه، شرط تمام شدن حد صبر یک پردازه را چک می‌کنیم و اگر سر رسیده باشد صفش را عوض می‌کنیم تا از starvation جلوگیری کنیم.



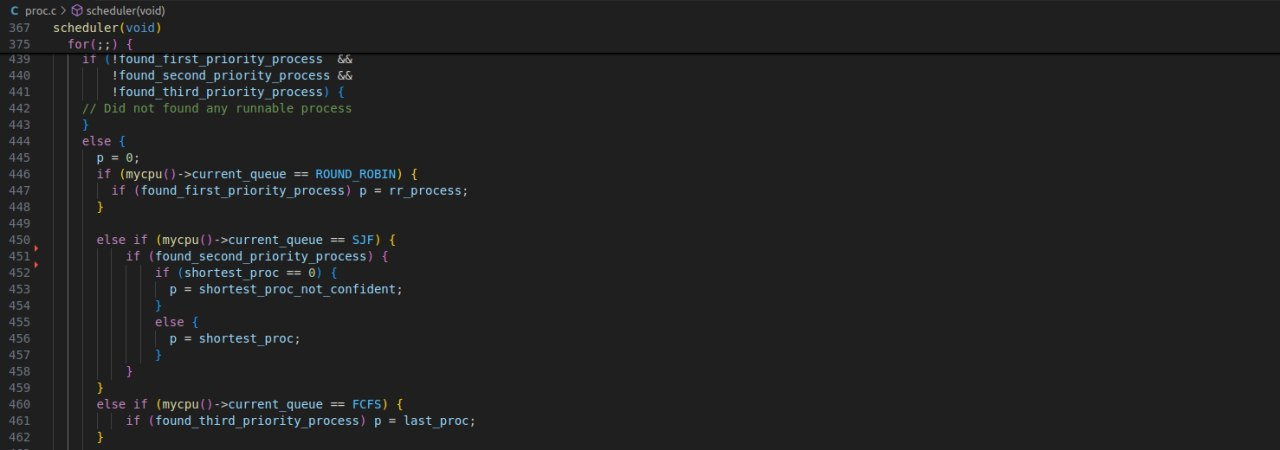
در این تابع اگر صف کنونی RR نباشد به صف پایین‌تر می‌رویم و مقداردهی متغیرهای مربوطه را نیز انجام می‌دهیم.



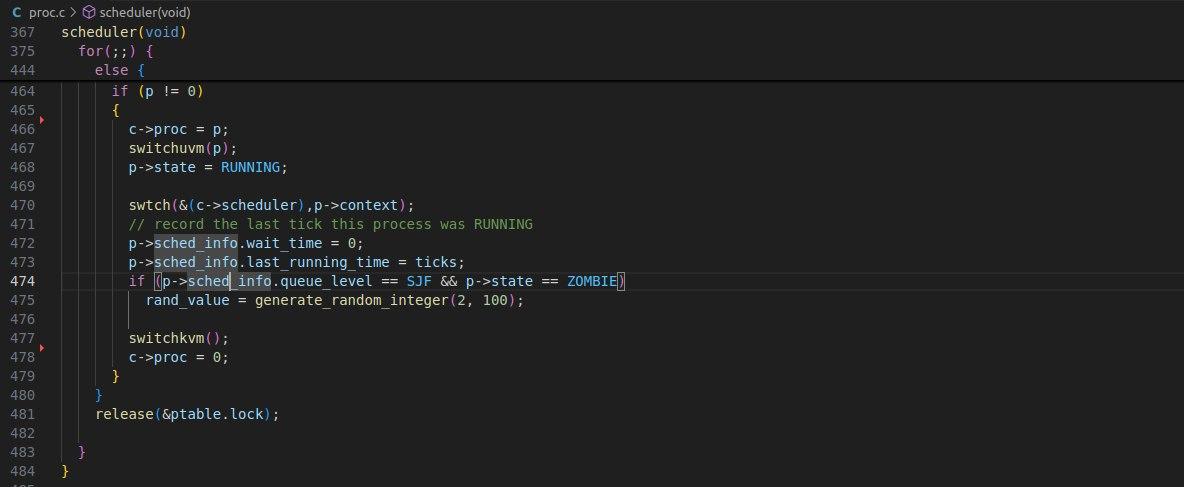
بعد از آن در ادامه حلقه روی پردازه‌ها، برای هر سه سیاست گفته شده پردازه انتخاب می‌کنیم. تا بعدا بر اسا صف کنونی cpu تصمیم به اجرای یک کدام از آن‌ها بگیریم.



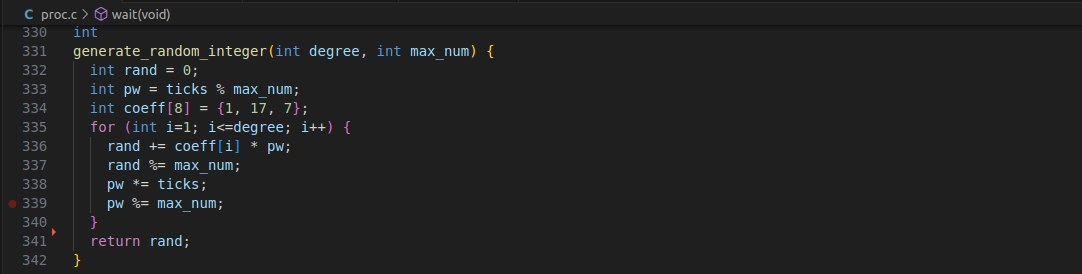
بعد از حلقه روی پردازه‌ها، باید بر اساس صف کنونی cpu یمی را انتخب کنیم. در ابتدا چک می‌کنیم آیا اصلا پردازه‌ای در ready queue قرار دارد یا خیر. اگر قرار داشت به ادامه عملیات می‌پردازیم. حالا یکی از پراسس‌ها را برای اجرا انتخاب می‌کنیم.



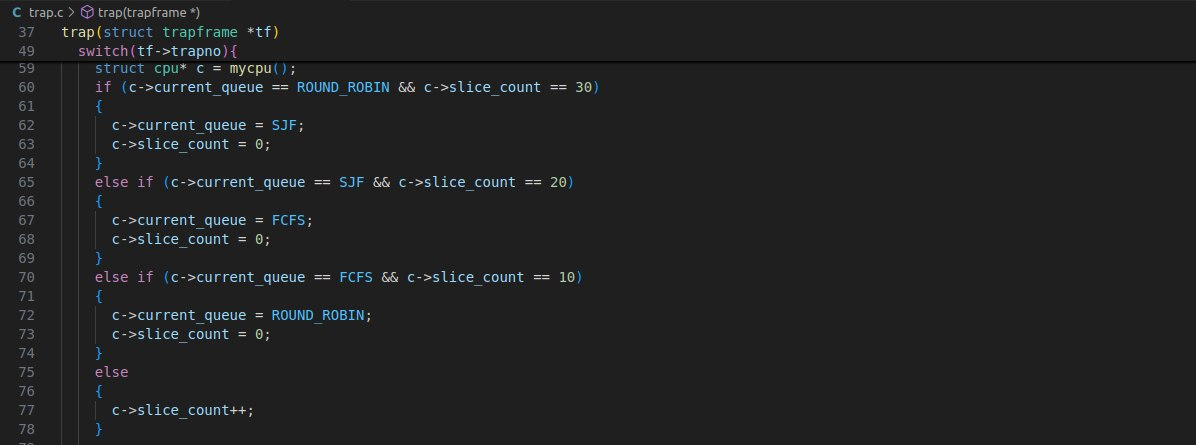
پردازه انتخاب شده اجرا می‌شود و فیلدهای مورد نیاز آپدیت می‌شوند. همچنین اگر در صف SJF بوده باشیم و پردازه کنونی تمام شود نیاز به تولید عدد رندوم جدیدی داریم.



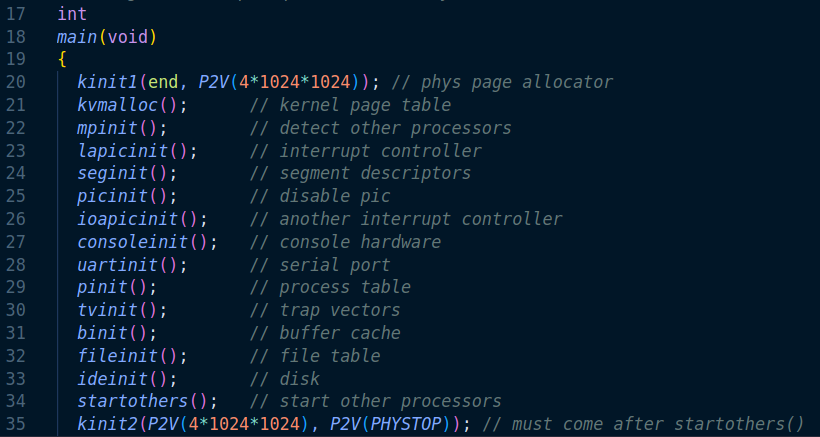
تابع تولید عدد رندوم بر اساس کلاک سیستم یک خروجی چندجمله‌ای می‌سازد که ضرایب آن اعداد اول هستند.



فرآیند time slicingدر فایل trap.c انجام گرفته. اگر شمارنده هر صف به حداکثر مقدار خود برسد آن را ریست کرده و به صف پایین‌تر می‌رویم، اگر نه شمارنده را به علاوه یک می‌کنیم.

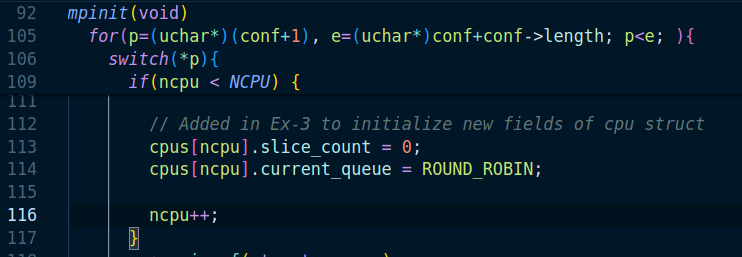


**20)**



با توجه به راهنمایی سوال به فایل main.c می رویم و مشاهده می کنیم که تابعی به نام mpinit صدا زده شده است که در آن کار initialize کردن انجام می گیرد.

این تابع در فایل mp.c قرار دارد لذا به آن فایل رفته و در تابع مربوط مقادیر جدید را اضافه می کنیم.



همان مواردی که به استراکت پردازنده اضافه کرده بودیم را اینجا مقدار اولیه می دهیم.

21)

به کمک بخش برش دهی زمانی ما کاری کردیم که لزوما تا وقتی صف اول پراسس دارد پردازنده روی آن نماند و به صف های دیگر هم اجازه اجرا بدهد.

یعنی در واقع ما گرسنگی بین صف ها را از بین بردیم اما مورد دیگری که با آن رو به رو هستیم گرسنگی بیت پراسس های هم صف است. یعنی مثلا یک پراسس در صف دو که کوتاه ترین کار اجرا می شود ممکن است هیچوقت نتواند پردازنده را بگیرد و اجرا شود لذا دچار گرسنگی می شود.

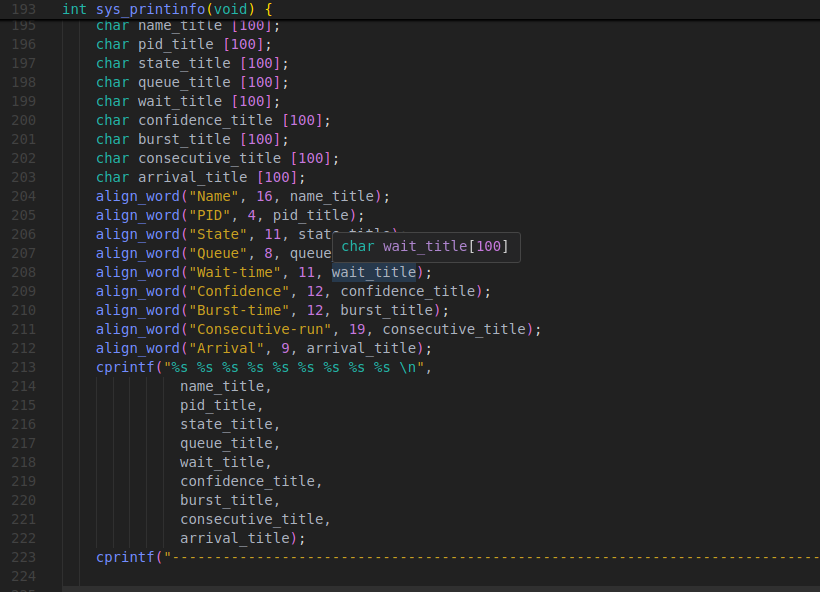
راه حل این موضوع ساز و کار سن است که در بخش بعد به آن می پردازیم و این مشکل را حل می کنیم.

**22)**

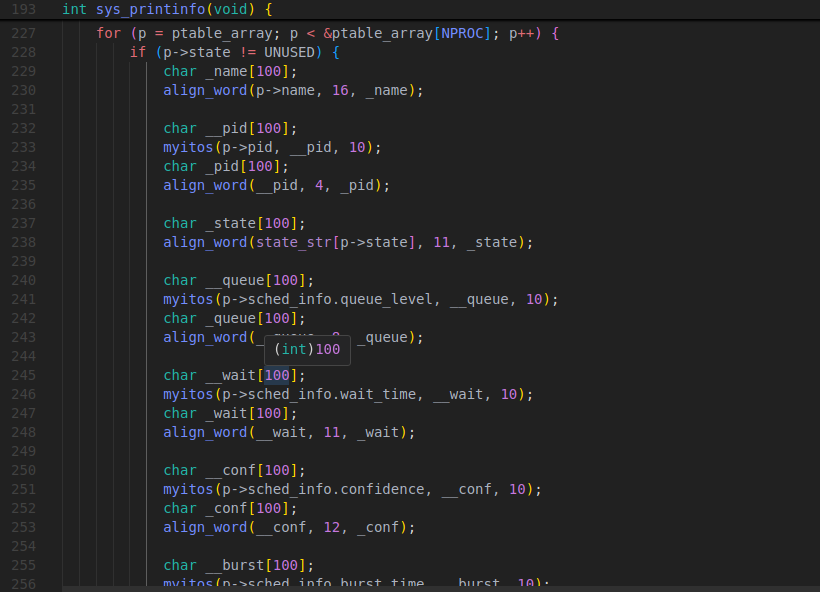
ممکن است یک process مدت زیادی منتظر عمل I/O و یا کار دیگری باشد که در این زمان حالت sleeping قرار دارد و منطقی نیست که جزو زمان‌های انتظار در صف ready در نظرش بگیریم. حتی ممکن است زمان انتظارش برای I/O از 800 کلاک بیشتر شود و ما در حالی اولویتش را بهبود بخشیم که کار قبلی‌اش تمام نشده.

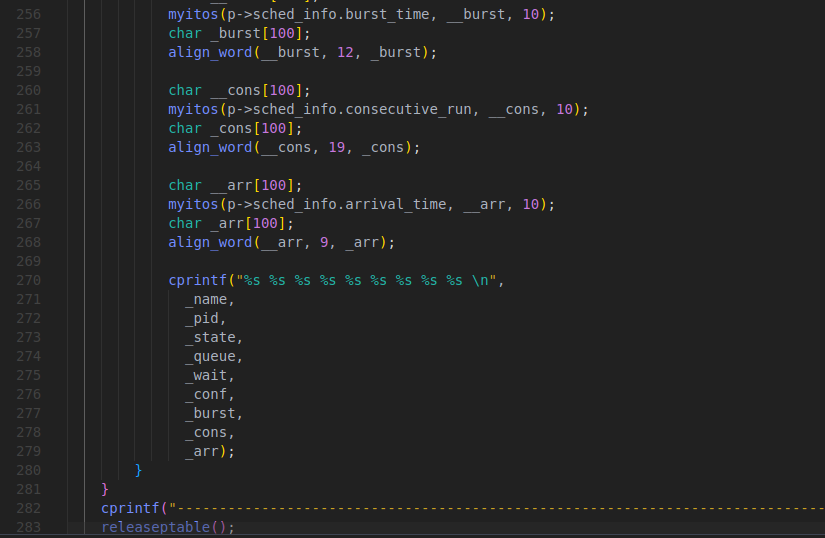
**فراخوانی های سیستمی مورد نیاز**

**Printinfo**

****

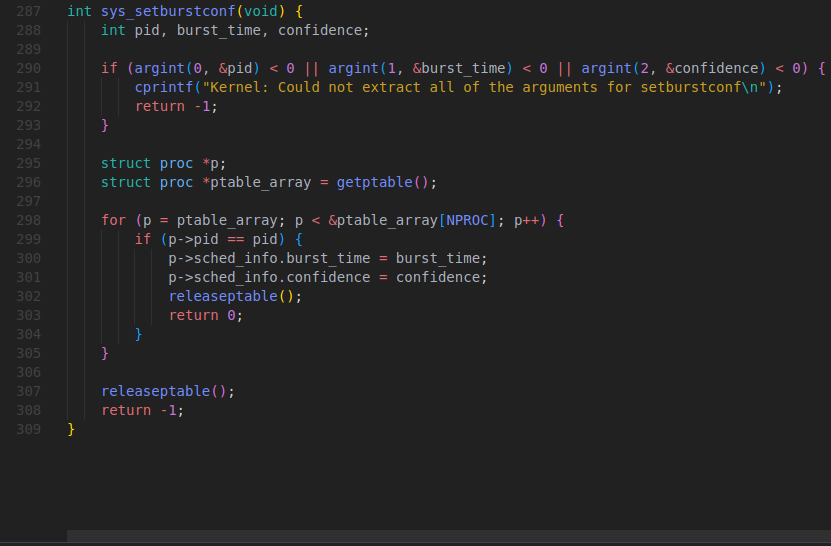
برای چاپ اطلاعات هم طبق خواسته صورت پروژه ، اطلاعات مربوطه را به صورت جدول پرینت میکنیم.

****

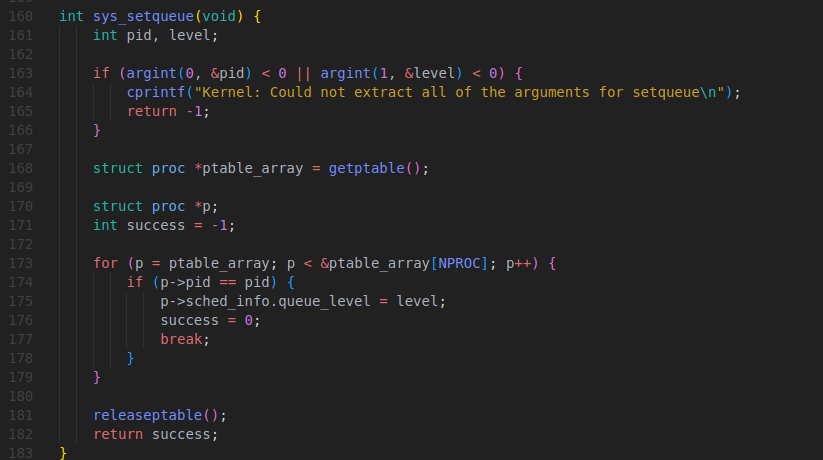
****

**SetBurstConf**

در این تابع هم pid مورد نظر را گرفته و confidence و burst time آن را به عدد های خواسته شده تغییر می دهیم .

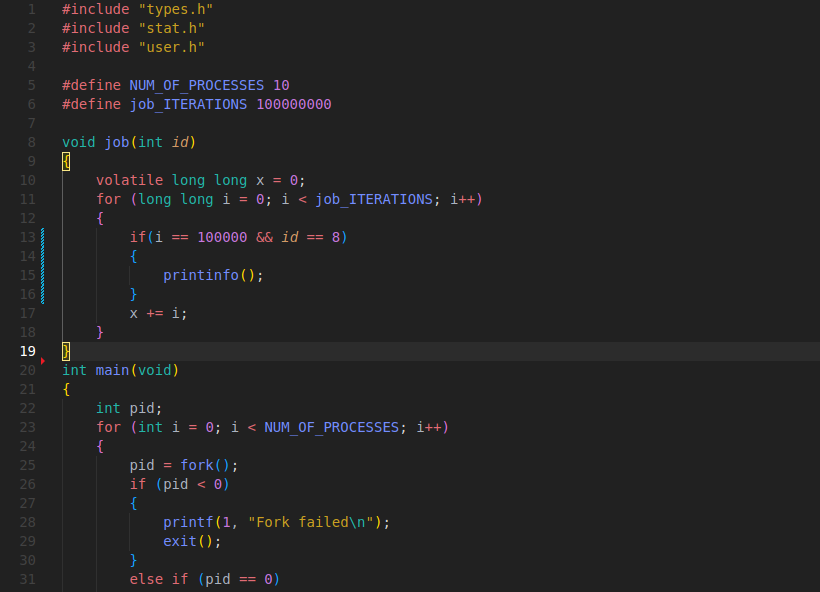
****

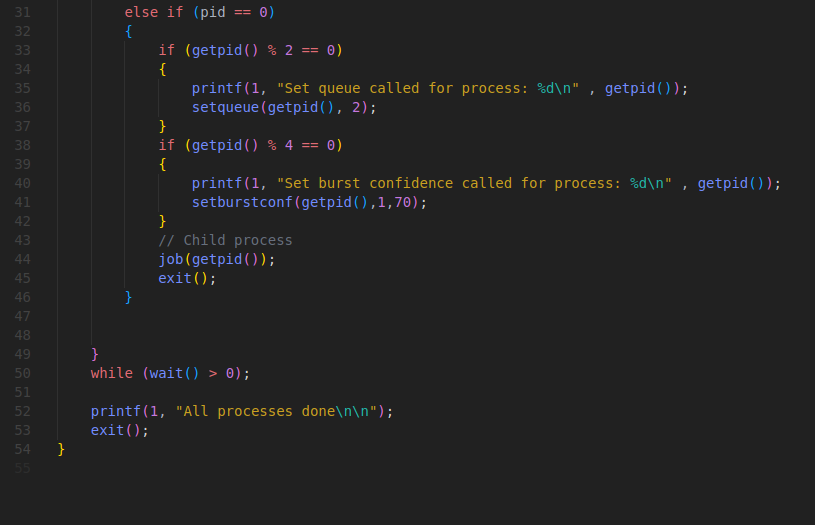
**SetQueue**

****

در تابع SetQueue هم pid مورد نظر را گرفته و صف پردازه را به صف مورد نظر تغییر میدهیم .

**Test\_systemcalls code**

****

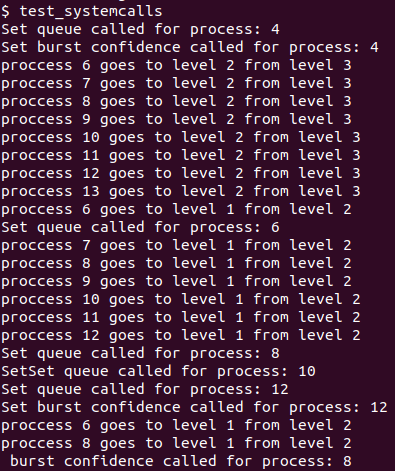
****

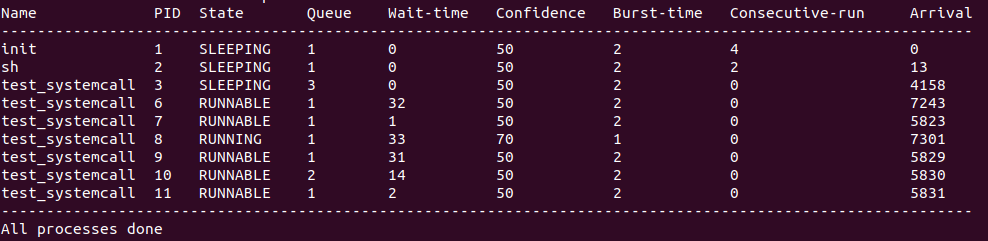
این کد در یک به تعداد دلخواه (اینجا 10 ) بار اجرا میشود و هر سری fork را صدا میزند و یک فعالیت طولانی برای هر پردازه در نظر میگیرد تا بتوانیم اعمال صف ها را به دقت بررسی کنیم . برای پردازه های زوج صف آنها را تغییر داده و به 2 منتقل میکنیم . برای پردازه های مضرب 4 burst time انها را از 2(دیفالت) به 1 تغییر داده و confidence آنها را نیز از 50 (دیفالت ) به 70 تغییر میدهیم .

**Test\_systemcalls result**

همان طور که توضیح داده شد ، پردازه های زوج از صف 3 به صف 2 منتقل شدند . و پردازه های مضرب 4 نیز burst time و confidence آنها دچار تغییر شد .(در جدول نهایی هم تغییر این مقادیر مشهود است . )

State, wait time , consecutive run , arrival time هم میتواند در هر بار اجرای برنامه متفاوت باشد ، پس صحبت خاصی درباره ی آنها نمی توان کرد . ولی مثلا مشاهده میکنیم که پردازه های در صف های مختلفی قرار دارند که نشان میدهد الگوریتم aging درست کار میکند و انتقال پردازه ها از صف با اولویت کمتر به صف با اولویت بیشتر به درستی انجام شده است .

****

****