**به نام خدا**

**گزارش کار پروژه سوم آزمایشگاه سیستم عامل**

**اعضا گروه :**

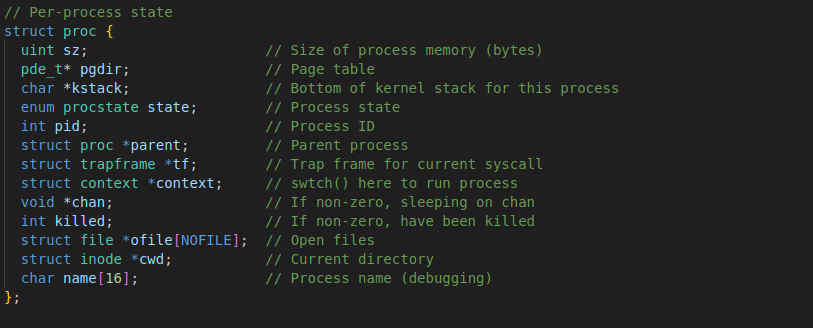
**مهراد لیویان 810101501**

**بهراد بینایی حقیقی 810101392**

**مرضیه موسوی کانی 810101526**

1و2-

ساختار pcb در xv6 :



در شکل کتاب pcb شامل بخش های زیر است:

Process state

Process number

Program counter

Register

Memory limit

List of open files

حال همانطور که می بینید ofile معادل list of open files است.state معادل process state است.pid معادل process number است.program counter توسط ساختار trapframe مدیریت می شود. Memory limit توسط متغیر sz مشخص می شود.

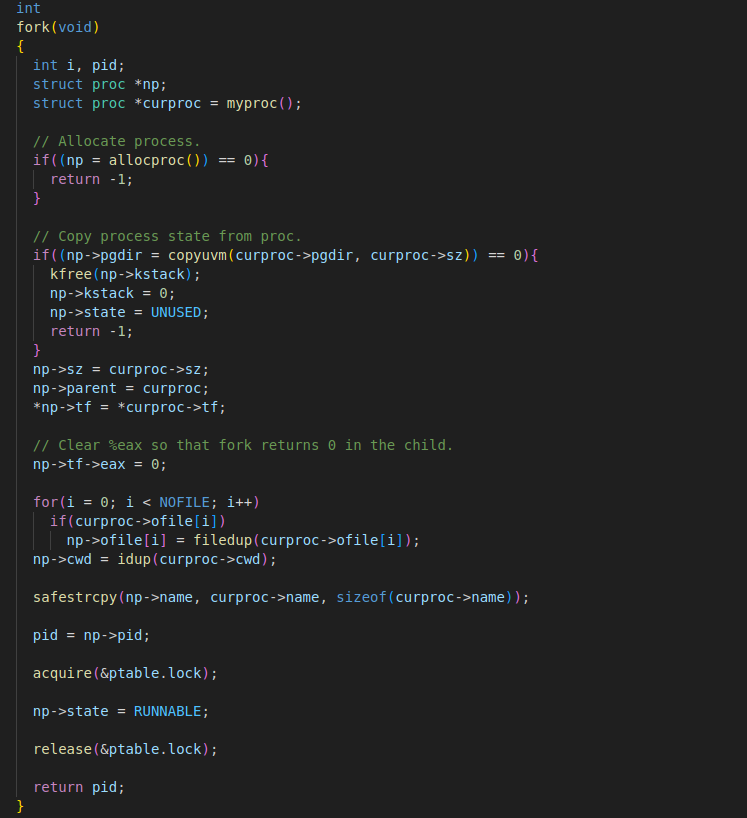
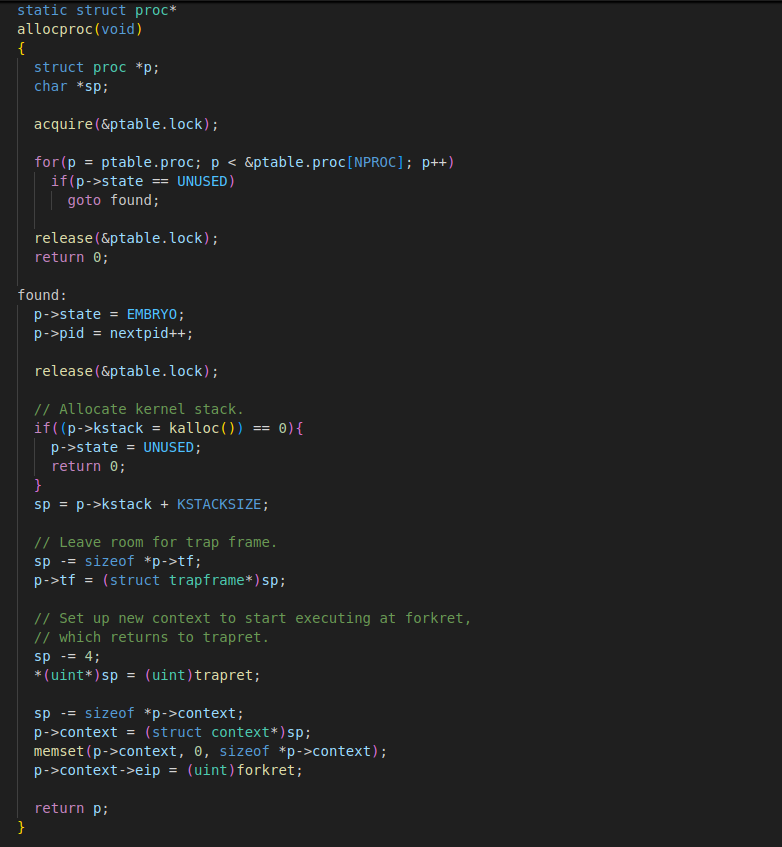
در فایل proc.h ، enum procstate را مشاهده می کنیم:



New معادل embryo می باشد.runnable یعنی پراسس هنوز ران نشده و قابل ران شدن است پس معادل ready است.sleepingمعادل waiting است.استیت running معادل هم نام خودش در مرجع است.استیت zombie یعنی پراسس ترمینیت شده ولی هنوز parent متوجه نشده است.unused یعنی اسلات پراسس خالی است.zombie را تا حدودی می توان معادل terminate گرفت.

3-

به ساختار دو تابع fork و allprocتوجه کنید:



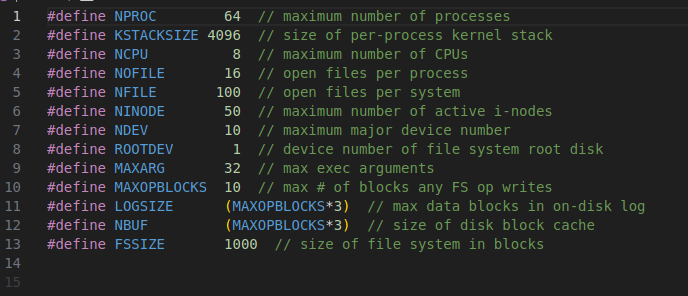
Allproc در fork صدا می شود.

در تابع allproc ابتدا ptable قفل می شود تا در هنگام تغییر ptable ، ptable تغییر نکند.سپس در ptable دنبال اسلات بلا استفاده می گردد.اگر پیدا شد اسلات را به پراسس جدید می دهد و وضیعت پراسس به embryo می رود.pid نیز آپدیت می شود.یک فضای کرنل استک به پراسس اختصاص می یابد.برای cpu context و trap frame نیز مقادیر دیفالت تعیین می شود.

حال در تابع fork پس از ساخت اولیه پراسس وضیعت به runnable معادل ready تغییر می کند.

4-

در فایل param.h داریم:



همانطور که از کامنت مشخص است سقف تعداد پردازه با کانستنت NPROC مشخص شده که 64 است.

اگر به کددallproc دقت کنید در صورتی که اسلات بلااستفاده یافت نشود تابع allproc عدد 0 را برمی گرداند که در تابعfork بعد از دریافت این عدد خودش-1 برمی گرداند که یعنی پراسس جدید ساخته نشده است.در موقع کد سطح کاربر ما این خطا را معمولا مدیریت می کنیم و اگر pid منفی یک بود یعنی پراسس جدید ساخته نشده است.

5-

در سیستم های چند پردازنده ای به دلیل مسائلی مانند race condition که ممکن است دیتاهای مهم و حساسی مانند وضیعت هر پردازه و اولویت پردازه و غیره توسط دو پراسس تغییر کند تناقضی پیش بیاید.همچنین در زمان تصمیم گیری برای انجام پردازه ی بعدی نباید در میانه ی کار جدول پردازه ها تغییر کند.

مشکلات زیر در صورت لاک نکردن ptable پیش بیاید:

**۱. تخصیص شناسه‌های پردازه و شکاف‌های جدول پردازه‌ها :**

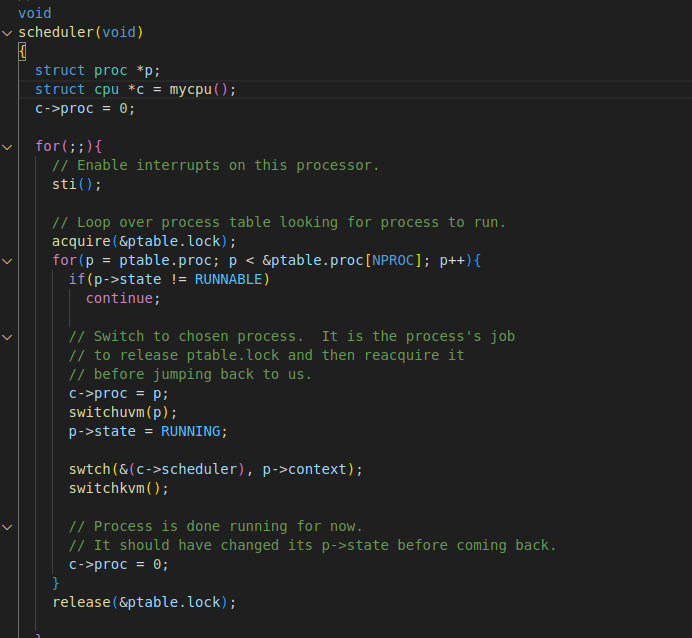
* **تخصیص یکتا و بدون تداخل:**  
  سیستم‌عامل اطمینان حاصل می‌کند که شناسه‌های پردازه (PID) و شکاف‌های مربوط به هر پردازه در جدول پردازه‌ها به‌طور یکتا و هماهنگ بین CPUها تخصیص داده شوند.  
  این کار از بروز مشکلاتی مانند تخصیص دوباره یک PID به دو پردازه مختلف یا تخصیص نادرست شکاف‌ها جلوگیری می‌کند.
* **سازگاری بین CPUها:**  
  در سیستم‌های چندپردازه‌ای و چند هسته‌ای، هماهنگی بین هسته‌ها برای تخصیص منابع بسیار حیاتی است. این هماهنگی از طریق قفل‌ها و مکانیزم‌های هم‌زمانی انجام می‌شود.

**۲. هم‌زمانی بین توابع exit و wait:**

* **مدیریت روابط والد و فرزند:**  
  هنگامی که یک پردازه خاتمه می‌یابد (exit)، پردازه والد (Parent) معمولاً از طریق تابع **wait()** منتظر می‌ماند تا وضعیت پردازه فرزند (Child) مشخص شود. قفل‌ها برای اطمینان از اینکه این روابط به درستی مدیریت شوند، استفاده می‌شوند.
* **جلوگیری از شرایط رقابتی:**  
  در صورتی که یک پردازه والد منتظر فرزند خود باشد و هم‌زمان پردازه فرزند خاتمه یابد، امکان بروز شرایط رقابتی وجود دارد. استفاده از قفل‌ها کمک می‌کند تا این شرایط مدیریت شود و والد بتواند به درستی وضعیت فرزند را بازیابی کند.
* **تمیزکاری منابع:**  
  هماهنگی بین این دو عملیات برای آزادسازی منابع پردازه (مانند حافظه و شکاف جدول) نیز حیاتی است.

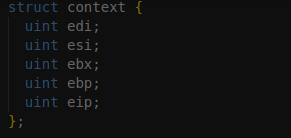
در سیستم های تک پردازنده ای به دلیل اجرای سریالی دستور ها لزومی به این کار در اغلب موارد نیست مگر اینکه interrupt بیاید و این interrupt ممکن است جدول پردازه ها را تغییر دهد بنابراین یا باید interrupt ها نافعال شوند یا جدول پردازه ها قفل شود.

6-



همانطور که در کد نیز مشخص است اگر پردازه runnable نباشد به iteration بعدی برای اجرا می رود.

7-



8-

رجیستر eip مخفف extended instruction pointer است .این رجیستر آدرس دستورالعمل بعدی را در کد برنامه نگهداری می‌کند. این رجیستر مشخص می‌کند که CPU در چه نقطه‌ای از برنامه باید به اجرای دستورالعمل‌ها ادامه دهد.

مقدار رجیستر های مهم ابتدا ذخیره می شوند و سپس به استک منتقل می شوند و در آخر در ساختار context کپی می شوند.

9-

بدیهتا زمانبندی پردازه ها مدیریت نمی شد.یک پردازه به طور نامحدود روی cpu اجرا می شد و باعث starvation بقیه پردازه ها می شد.

از آن طرف به طور کلی فعال نشدن وقفه باعث می شود که مدیریت منابع مشترک مدیریت ورودی خروجی ها و .. به درستی انجام نگیرد.

10-

پس از انجام عملیات گفته شده در هر 10 ثانیه حدود 1000 عدد چاپ شد.بنابراین هر 10 میلی ثانیه وقفه ی تایمر صادر می شود.

11-

تابع yield باعث انجام شدن گذار interrupt می شود.این تابع باعث تغییر حالت پراسس فعلی به runnable و آزاد کردن پردازنده باعث می شود پردازنده امکان رسیدن به وقفه ها یا انتخاب پراسس جدید برای scheduler می شود.

12-

با توجه به آن که هر وقفه ی تایمر نشان دهنده یک کوانتوم زمانی می باشد بنابراین هر کوانتوم زمانی 10 میلی ثانیه است.

13-

در سیستم‌عامل xv6، تابع wait برای منتظر ماندن تا اتمام کار یک پردازه‌ی فرزند استفاده می‌شود. این تابع در نهایت از مکانیزمی برای انتظار استفاده می‌کند تا زمانی که وضعیت پردازه‌ی فرزند zombie تغییر کند.

با قفل کردن ، تمام پردازه‌های فرزند پردازه جاری بررسی می‌شوند. اگر پردازه‌ای با وضعیت zombie یافت شود، منابع مربوط به آن آزاد شده و پردازه از جدول پردازه‌ها حذف می‌شود.اگر هیچ پردازه‌ی فرزندی با وضعیت zombieیافت نشود، تابع sleepفراخوانی می‌شود. تابع sleepپردازه جاری را به حالت sleeping منتقل می‌کند و منتظر می‌ماند تا وضعیت پردازه فرزند تغییر کند.

14-

انتظار برای رویدادها،همگام‌سازی،مدیریت منابع،ورودی/خروجی،زمان‌بندی،و مدیریت پردازه‌ها

15-

تابع wakeup نقش کلیدی در بیدار کردن پراسس هایی دارد که به حالت sleeping رفته اند.

16-

باعث تغییر وضیعت از وضیعت sleeping به runnable می شود.

17-

در سیستم عامل **xv6**، تابع دیگری نیز وجود دارد که می‌تواند باعث انتقال پردازه از حالت **SLEEPING** به حالت **RUNNABLE** شود. این تابع،**kill()** است.

**توضیح عملکرد تابع kill :**

**ارسال سیگنال به پردازه:**

تابع **kill()** برای ارسال سیگنال به یک پردازه استفاده می‌شود. این سیگنال معمولاً برای خاتمه دادن به پردازه به کار می‌رود.

**انتقال پردازه از SLEEPING به RUNNABLE پ:**

اگر پردازه‌ای در حالت **SLEEPING** باشد (در انتظار یک رویداد یا منبع)، تابع **kill()** این پردازه را مجبور به بیدار شدن می‌کند.

این انتقال، پردازه را از حالت**SLEEPING**  به **RUNNABLE** می‌برد، به این معنی که پردازه برای اجرا آماده می‌شود و در صف آماده (Ready Queue) قرار می‌گیرد.

**پاسخ به سیگنال:**

این رفتار باعث می‌شود که حتی پردازه‌هایی که در حالت خواب هستند، بتوانند به سیگنال‌های سیستم (مانند سیگنال خاتمه) پاسخ دهند.

18-

در سیستم‌عامل **xv6**، پردازه‌های یتیم (Orphan Processes) پردازه‌هایی هستند که والدین خود را از دست داده‌اند، یعنی والدین آن‌ها خاتمه یافته‌اند یا به دلایلی دیگر قادر به مدیریت آن‌ها نیستند. برای جلوگیری از مشکلاتی مانند *زومبی پروسس‌ها* یا عدم مدیریت این پردازه‌ها، **xv6** رویکرد زیر را در پیش می‌گیرد:

**انتقال پردازه‌های یتیم به پردازه init:**  
در xv6، اگر پردازه‌ای والد خود را از دست بدهد، سیستم‌عامل والد این پردازه را به پردازه **init** می دهد. پردازه init یک پردازه ویژه است که همیشه در حال اجرا بوده و مسئول رسیدگی به این پردازه‌ها است.

**(Waiting) توسط پردازه init:**  
پردازه init به طور مداوم پردازه‌های فرزند خود را بررسی می‌کند و زمانی که این پردازه‌ها به پایان برسند، منابع آن‌ها را آزاد می‌کند. این رویکرد تضمین می‌کند که هیچ پردازه یتیمی به وضعیت زامبی نرود.

**حذف منابع و جلوگیری از نشتی:**  
وقتی یک پردازه یتیم به پایان می‌رسد، پردازه init وظیفه دارد با سیستم کال waitجدول پردازه‌ها و دیگر منابع مربوط به آن را پاک کند تا از هرگونه نشتی منابع جلوگیری شود.

**مزایای این رویکرد:**

**مدیریت ساده‌تر:** پردازه init به عنوان یک نقطه مرکزی برای مدیریت تمام پردازه‌های یتیم عمل می‌کند.

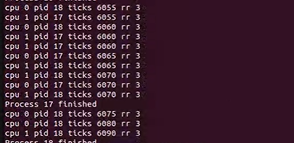
**پایداری سیستم:** منابع مربوط به پردازه‌ها به درستی آزاد شده و مانع از کند شدن یا خرابی سیستم می‌شود.

**کاهش بار مدیریت والد:** پردازه‌های والد نیازی به بررسی دقیق وضعیت فرزندان ندارند.

این رویکرد الهام گرفته از رفتار مشابه در سیستم‌عامل **UNIX** است و کارایی و سادگی مدیریت پردازه‌ها را تضمین می‌کند.

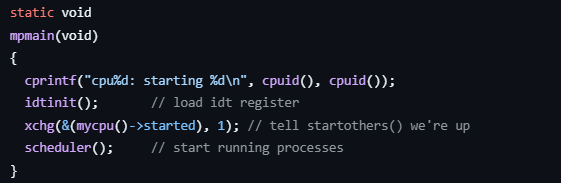
19-

ترتیب حفظ می شود ولی پردازه ها بین دو core تقسیم می شوند.



20-

**دلایل انتخاب mpmain:**  
تابع mpmain بعد از مقداردهی اولیه سیستم و شروع کار هر پردازنده (CPU) فراخوانی می‌شود. این مکان جایی است که پردازنده‌ها آماده انجام وظایف خود هستند، و ساختار cpu نیز در این مرحله آماده مقداردهی است.  
در mpmain، مقداردهی اولیه به ساختار مربوط به CPU انجام می‌شود. این تابع برای هر پردازنده به طور جداگانه اجرا می‌شود و به مقداردهی دقیق و مستقل هر پردازنده کمک می‌کند.



در صورتی که مقداردهی اولیه باید زودتر انجام بگیرد،می شود در تابع cpuinit نیز می توان این کار را کرد.

21-

در سطح دوم، از الگوریتم **Shortest Job First (SJF)** استفاده می‌شود. این الگوریتم به پردازه‌هایی که زمان اجرای کوتاه‌تری دارند، اولویت بیشتری می‌دهد.  
حالا فرض کنید یک پردازه با **زمان اجرای طولانی (Long Burst Time)** در این صف قرار دارد. به‌طور هم‌زمان، پردازه‌های زیادی با زمان‌های اجرای کوتاه‌تر به صف اضافه می‌شوند.

**نتیجه:**

* هر بار که سیستم زمان‌بند می‌خواهد پردازه‌ای را اجرا کند، پردازه‌ای با زمان اجرای کوتاه‌تر از پردازه طولانی‌تر انتخاب می‌شود.
* این روند باعث می‌شود پردازه با زمان طولانی هرگز به CPU دسترسی پیدا نکند.
* در نهایت، این پردازه ممکن است برای مدت زمان نامحدود در صف بماند و به **گرسنگی** دچار شود.

**چرا این مشکل در SJF رخ می‌دهد؟**

SJF ذاتاً پردازه‌های کوتاه‌تر را ترجیح می‌دهد و به پردازه‌های طولانی‌تر توجه کمتری دارد. حتی اگر از روش‌های زمان‌بندی وزنی مانند **(WRR)**برای تنظیم زمان‌بندی استفاده شود، این مشکل حل نمی‌شود، زیرا WRR تأثیری بر رفتار SJF ندارد.

**سناریوی گرسنگی در سطح سوم :**

در سطح سوم، از الگوریتم **First Come, First Served** استفاده می‌شود. این الگوریتم پردازه‌ها را بر اساس ترتیب ورودشان به صف اجرا می‌کند. حالا فرض کنید پردازه‌ای وارد صف شود که دارای یک حلقه بی‌نهایت است، به طوری که در زمان اجرای خود **Interrupt** ایجاد نمی‌کند.

**نتیجه:**

* پردازه با حلقه بی‌نهایت برای مدت نامحدود CPU را اشغال می‌کند.
* هیچ پردازه دیگری در این سطح نمی‌تواند اجرا شود، زیرا در FCFS، پردازه‌ها تا زمانی که کارشان تمام نشود CPU را آزاد نمی‌کنند.
* این رفتار باعث می‌شود سایر پردازه‌ها در همان سطح دچار **گرسنگی** شوند.

**چرا این مشکل در FCFS رخ می‌دهد؟**  
 FCFS به گونه‌ای طراحی شده است که پردازه‌ای که زودتر وارد صف شود، بدون هیچ وقفه‌ای تا پایان کارش اجرا می‌شود. برخلاف سایر الگوریتم‌ها مانند Round Robin، در اینجا هیچ مکانیزمی برای محدود کردن زمان اجرای یک پردازه وجود ندارد.

**چرا این سناریوها در سطح اول رخ نمی‌دهند؟**

در سطح اول، از الگوریتم **Round Robin** استفاده می‌شود. در این روش، هر پردازه دارای یک بازه زمانی مشخص است. پس از اتمام این بازه:

1. پردازه از CPU خارج شده و به انتهای صف منتقل می‌شود.
2. نوبت به پردازه بعدی در صف می‌رسد.

حتی اگر یک پردازه دارای یک حلقه بی‌نهایت باشد، نمی‌تواند CPU را برای مدت طولانی در اختیار داشته باشد، زیرا کوانتوم آن تمام شده و به انتهای صف فرستاده می‌شود. این روش تضمین می‌کند که هیچ پردازه‌ای برای مدت طولانی منتظر نماند و **گرسنگی** رخ ندهد.

22-

مدت زمانی که پردازه در وضعیت **SLEEPING** قرار دارد، به‌عنوان زمان انتظار پردازه از منظر زمان‌بندی در نظر گرفته نمی‌شود، زیرا این وضعیت به رقابت برای استفاده از CPU مربوط نیست. اضافه کردن زمان خواب (SLEEPING) به زمان انتظار می‌تواند معیارهای زمان‌بندی را مخدوش کند و مشکلاتی را در سیاست‌های زمان‌بندی ایجاد کند. دلایل این موضوع به شرح زیر است:

**۱. ماهیت وضعیت SLEEPING**

* زمانی که پردازه در وضعیت **SLEEPING**  قرار دارد، منتظر وقوع یک رویداد خارجی (مانند تکمیل یک عملیات ورودی/خروجی، دریافت سیگنال، یا آزاد شدن قفل) است و نیازی به CPU ندارد.
* در این حالت، پردازه به‌طور فعال در**(Ready Queue)**  حضور ندارد و در رقابت برای منابع CPU شرکت نمی‌کند.

**۲. تعریف دقیق زمان انتظار (Waiting Time)**

* **زمان انتظار پردازه** معیاری است که مدت زمانی را که یک پردازه منتظر اجرا توسط CPU بوده اندازه‌گیری می‌کند.
* در حالت SLEEPING، پردازه در انتظار منابع دیگری مانند دیسک یا شبکه است، نه CPU. در نتیجه، این زمان نباید به عنوان بخشی از زمان انتظار پردازه برای CPU محاسبه شود.

**۳. اثر مخدوش کردن زمان انتظار**

اگر زمان SLEEPING به زمان انتظار پردازه اضافه شود، این موضوع می‌تواند مشکلات زیر را ایجاد کند:

1. **اولویت‌های نادرست:**  
   پردازه‌هایی که بیشتر زمان خود را در حالت SLEEPING سپری می‌کنند، ممکن است به اشتباه به‌عنوان پردازه‌هایی با زمان انتظار طولانی در نظر گرفته شوند. این امر می‌تواند باعث شود که پردازه‌هایی که نیازی به CPU ندارند، **اولویت بیشتری** دریافت کنند یا به طور ناعادلانه از زمان‌بندی کنار گذاشته شوند.
2. **تصویر نادرست از عملکرد سیستم:**  
   زمان SLEEPING ارتباطی با رقابت برای CPU ندارد. اگر این زمان به عنوان زمان انتظار محاسبه شود، عملکرد واقعی الگوریتم‌های زمان‌بندی به‌درستی ارزیابی نخواهد شد.

**۴. تأثیر بر الگوریتم‌های حساس به زمان انتظار**

برخی الگوریتم‌های زمان‌بندی، مانند **Shortest Job First (SJF)** یا **Priority Scheduling**، برای تصمیم‌گیری‌های خود به محاسبات دقیق زمان انتظار متکی هستند.

* اگر زمان SLEEPING به زمان انتظار اضافه شود، پردازه‌هایی که در صف آماده (Ready Queue) حضور ندارند، به اشتباه به عنوان پردازه‌های با انتظار طولانی در نظر گرفته می‌شوند.
* این موضوع می‌تواند باعث شود که پردازه‌های SLEEPING اولویت بیشتری پیدا کنند، در حالی که عملاً نیازی به اجرای فوری ندارند.

5. **تعریف دقیق consecutive time** :

مجموع تیک های سپری شده هر پراسس در استیت رانینگ .

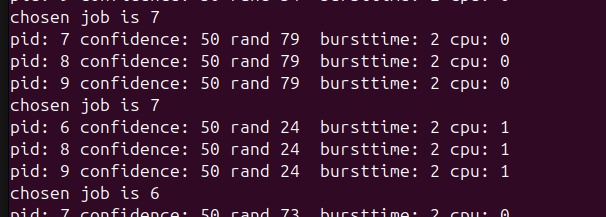
**تست صف ها و تست کلی برنامه:**

**Sjf:**

روی یک cpu:

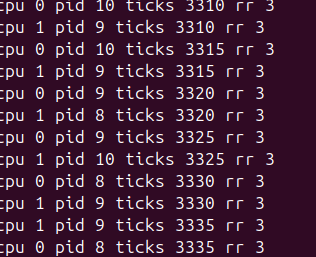
همانطور که در این بخش می بینید با توجه به آنکه عدد رندوم از confidence همه ی پراسس ها کوچکتر است همه ی پراسس ها ولید هستند و کوچک ترین bursttime که برابر 2 است را انتخاب می کنیم.

روی هر دو cpu :

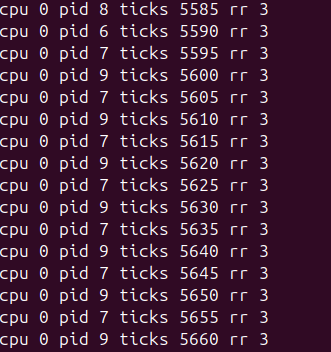


**Roundrobin:**

با 2 cpu:



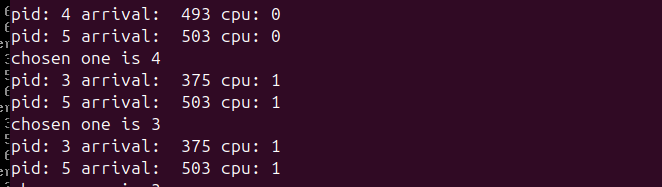
با تک cpu :



همانطور که می بینید در هر دو حالت درست کار می کند.

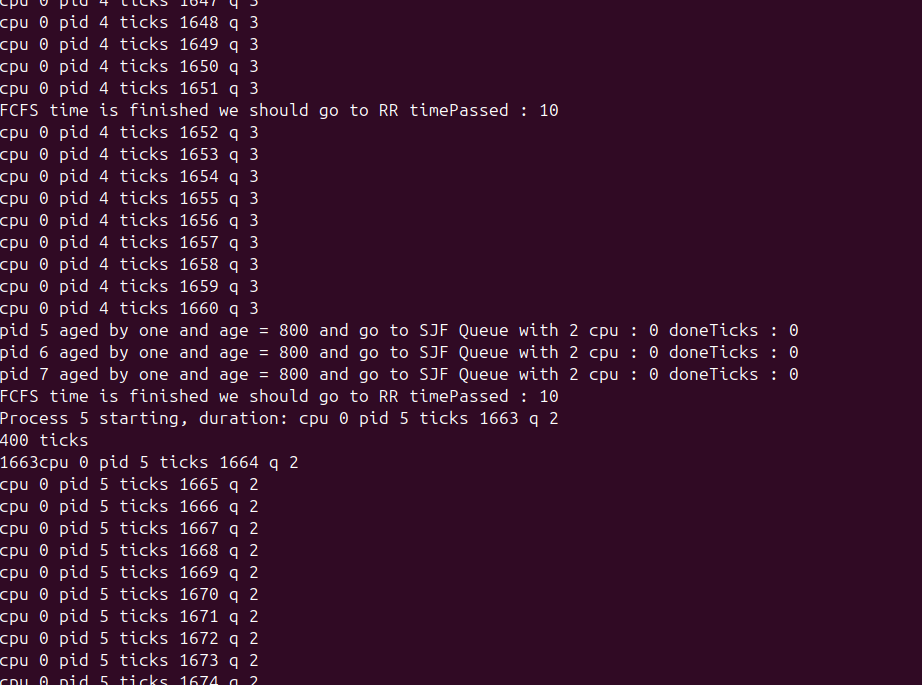
**FCFS:**

روی دو هسته:



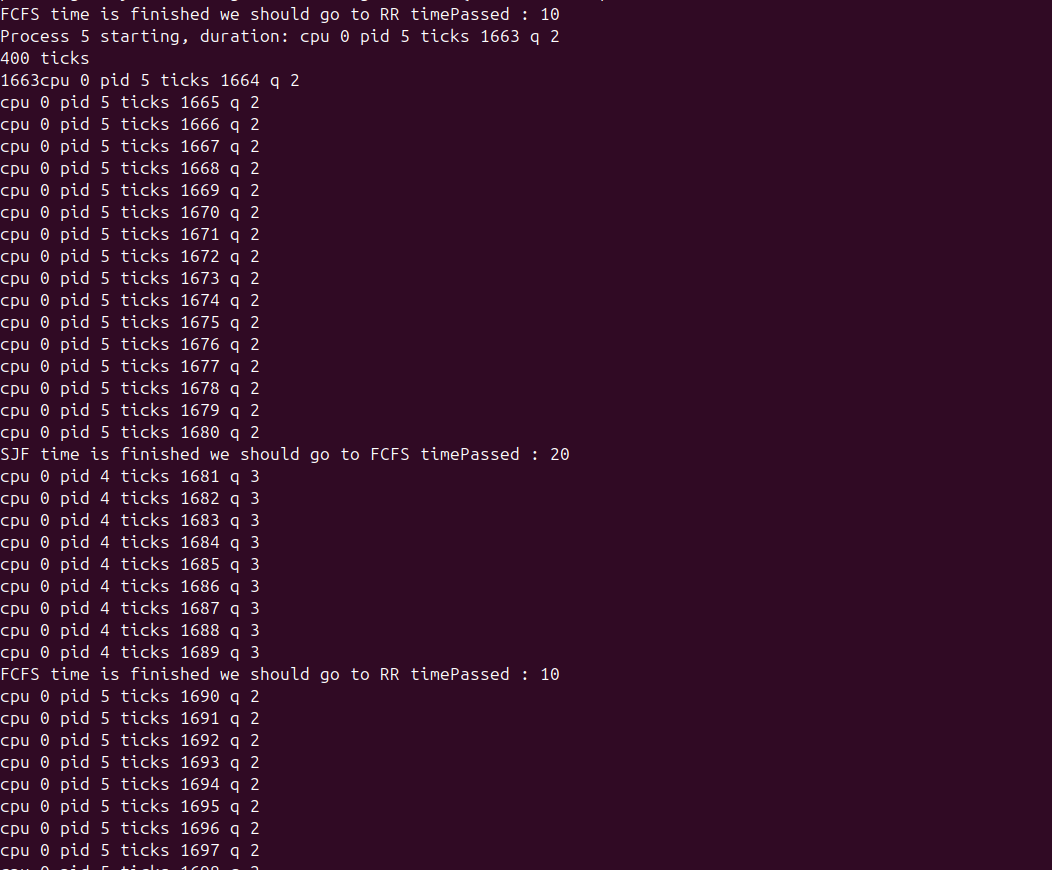
**تست aging :**

بعد از گذشت 800 تیک به پردازه هایی که در حالت runable هستند aging وارد می کنیم تا یه صف بالا تر روند :

****

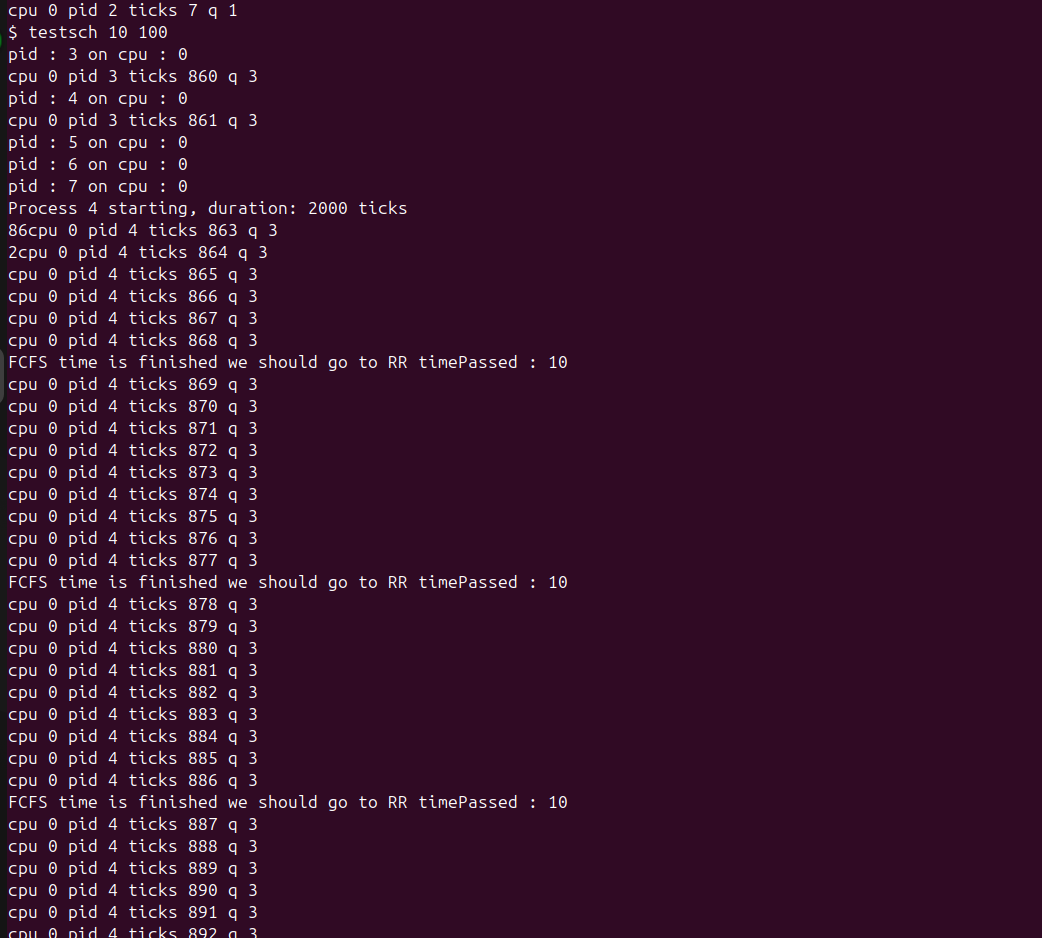
**تست aging و time slicing همزمان :**

در این تصویر هم مشخص است بعد از انجام شدن aging بین دو صف جا به جا می شویم :

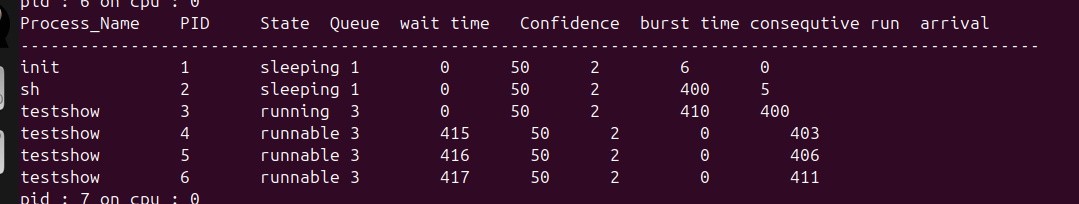
****

**تست time slicing :**

**همانطور که در تصویر واضح است هر ده تیک برای صف سوم که FCFS هست داریم تایم اسلایسینگ انجام می دهیم :**

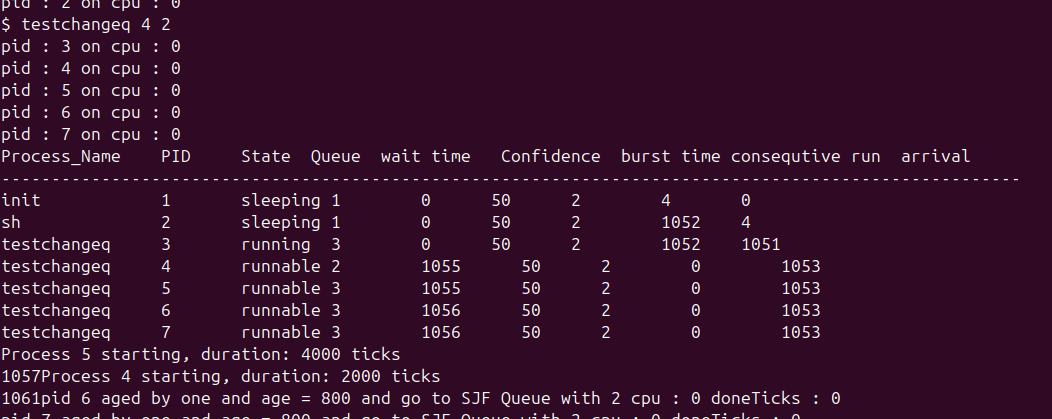
****

**تست سیستم کال چاپ اطلاعات :**

****

**تست تغییر صف :**

پردازه 4 را به صف دوم انتقال می دهیم :



**تست دادن پارامتر های SJF :**

مقدار burst time را برای پردازه های فرزند برابر 33 و مقدار confidence را برای پردازه های فرزند برابر 60 قرار دادیم که نتایج زیر حاصل شد :

