به نام خدا

گزارشکار آزمایش سه آزمایشگاه سیستم عامل

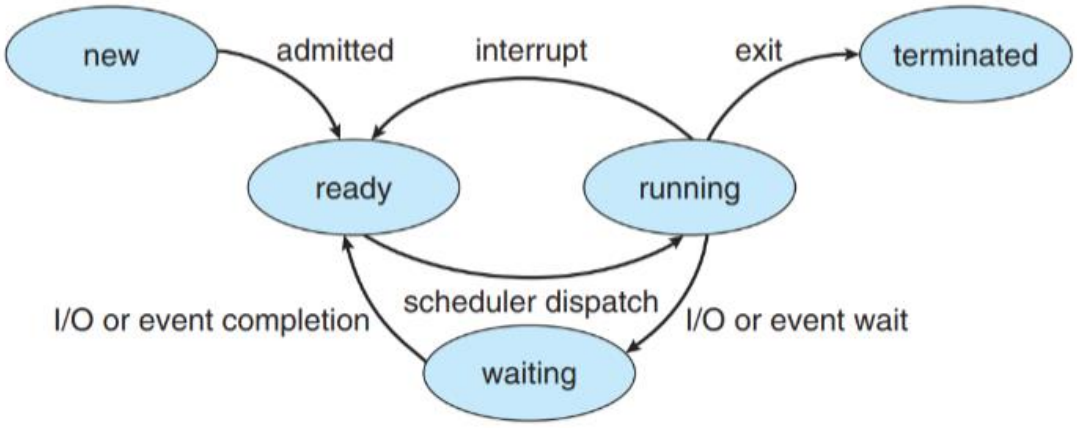
اعضای گروه:

علی زمانی 810101436

برنا فروهری 810101480

نیما خنجری 810101416

پرسش 1: ساختار PCB و همچنین وضعیتهای 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؟ (ذکر حداقل ۵ مورد و معادل آنها در xv6)



شکل ۱. چرخهٔ وضعیت یک پردازه

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

می توان دید که ساختار PCB در xv6 به صورت ساختار struct proc تعریف شده است. یکی ازدیتا هایی که در PCB موجود می باشد وضعیت پردازه است.

وضعیت های مختلفی که در سیستم عامل xv6 برای یک پردازه تعریف شده به این ترتیب هستند:

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

1- UNUSED : این وضعیت معادل وضعیت terminated می باشد یعنی نماینده حالتی است که پردازه خاتمه یافته یا اصلا هنوز ایجاد نشده است.

2- EMBRYO : این وضعیت نماینده حالتی است که پردازه تازه ایجاد شده ولی هنوز آماده اجرا نیست.(عملا مشابه وضعیت new در شکل می باشد)

3- RUNNABLE : اگر پردازه در این وضعیت باشد یعنی آماده اجرا است و صرفا باید منتظر بماند تا به cpu اختصاص یابد. می توان گفت این وضعیت در xv6 معادل وضعیت ready است.

4- RUNNING : در این وضعیت پردازه در حال اجرا شدن روی cpu می باشد.مشابه running در شکل بالا.

5- SLEEPING : در این وضعیت پردازه منتظر تکمیل شدن یک فرآیند I/O یا رویدادی خاص است و عملا معادل وضعیت waiting در شکل بالا می باشد.

6- ZOMBIE : این وضعیت نماینده حالتی است که پردازه موردنظر خاتمه پیدا کرده ولی هنوز resource های آن به طور کامل آزاد نشده اند.

داده های موجود در ساختار به تصویر کشیده شده در شکل 3.3 منبع درس و معادل آن ها در سیستم عامل xv6:

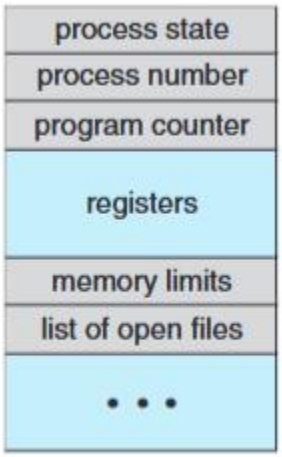


Figure 3.3 Process control block (PCB).

1- Process State : در بخش بالا کامل مشابهت هایی که این بخش در شکل 3.3 با وضعیت های یک پردازه در xv6 دارند را توضیح دادیم.

2- Process Number : این بخش معادل pid در xv6 است که id برای هر پردازه را ذخیره می کند.

3- Program Counter : این بخش که همان شمارنده برنامه است اشاره گری به آدرس دستور بعدی ای است که باید اجرا شود. در xv6 ساختاری داریم به نام context که شامل رجیستر های cpu می باشد. این بخش یعنی شمارنده برنامه هم در xv6 در این ساختار context ذخیره می شود.

4- Registers : همانطور که در بخش قبلی هم اشاره ای داشتیم معادل این بخش در xv6 ساختار context می باشد.

5- List of Open Files : در شکل این بخش نماینده فایل هایی است که پردازه موردنظر به آن ها دسترسی دارد. در xv6 آرایه ofile[] را در PCB داریم که برای هر پردازه لیست فایل هایی که پردازه موردنظر به آن ها دسترسی دارد را ذخیره می کند.

پرسش 2:هر کدام از وضعیتهای تعریف شده معادل کدام وضعیت در شکل ۱ میباشند؟

در بخش ابتدایی پاسخ پرسش 1 به این سوال جواب دادیم.

پرسش 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 ۱ به تصویر کشیده شده خواهد شد؟ وضعیت یک پردازه در xv6 در این گذار از چه حالتی حالتهایی به چه حالت حالتهایی تغییر میکند؟ پاسخ خود را با پاسخ سوال ۲ مقایسه کنید.

در فایل proc.c تابعی به نام allocproc وجود دارد که پردازه جدیدی ایجاد کرده و آن را مقداردهی اولیه می‌کند. در ابتدای ایجاد یک پردازه, پردازه موردنظر در وضعیت UNUSED می باشد اما در تابع allocproc وضعیت آن به EMBRYO تغییر می‌یابد. همانطور که در سوال های بخش قبل دیدیم, این وضعیت نشان دهنده آن است که پردازه موردنظر با وجود این که ایجاد شده هنوز آماده اجرا شدن نیست.

بعد از این مرحله, با استفاده از تابعی به نام userinit مقداردهی‌های اولیه از جمله مقداردهی program counter انجام شده و اولین پردازه ایجاد می شود. پردازه مورد نظر ما در همین تابع یا در هنگام اجرای پردازه‌های جدید در توابعی مثل fork از وضعیت EMBRYO به وضعیت RUNNABLE میرسد. توجه داشته باشید که وضعیت RUNNABLE معادل همان وضعیت Ready در شکل می باشد.

این گونه پردازه از وضعیت UNUSED به EMBRYO و در نهایت به RUNNABLE می رسد.این تغییر وضعیت معادل انجام گذار از حالت new به ready است.

پرسش 4: سقف تعداد پردازه های ممکن در xv6 چه عددی است؟ در صورتی که یک پردازه تعداد زیادی پردازه فرزند ایجاد کند و از این سقف عبور ،کند کرنل چه واکنشی نشان داده و برنامه سطح کاربر چه بازخوردی دریافت میکند؟

درxv6 ، سقف تعداد پردازه‌هایی که سیستم می‌تواند همزمان مدیریت کند، با استفاده از ثابتی به نام NPROC که در فایل param.h تعریف شده است, مشخص میشود. مقدار پیش‌فرض این ثابت معمولاً برابر64 است. این بدان معناست که xv6 ماکسیمم می‌تواند 64 پردازه فعال را همزمان مدیریت کند.

```
#define NPROC        64  // maximum number of processes
```

حال اگر یک پردازنده تعداد زیادی پردازه فرزند ایجاد کند به طوری که تعداد پردازه‌ها از این مقدار تعیین شده بیشتر شوند, کرنل در هنگام فراخوانی تابع fork، نمی‌تواند پردازه جدیدی تخصیص دهد. به همین دلیل, تابع allocproc که مسؤل ایجاد پردازه های جدید است return NULL خواهد کرد. این امر باعث خواهد شد که فراخوانی fork در برنامه سطح کاربر شکست بخورد. در نتیجه تابع fork در برنامه سطح کاربر مقدار-1 را برمی‌گرداند. همانطور که از آزمایش های قبلی می دانیم, این مقدار به معنی آن است که ایجاد پردازه جدید موفق نبوده. پس برنامه سطح کاربر می‌تواند با بررسی مقدار بازگشتی از fork، این خطا را شناسایی کرده و ارور مناسب را برای یوزر نمایش دهد.

پرسش 5: چرا نیاز است در ابتدای هر حلقه تابع scheduler جدول پردازه ها قفل شود؟ آیا در سیستمهای تک پردازه ای هم نیاز است این کار صورت بگیرد؟

در سیستم‌عاملxv6 ، ptable داده‌ساختاری مشترک است که اطلاعات مربوط به تمام پردازه‌های سیستم را نگهداری می‌کند. از آنجایی که xv6 می‌تواند بر روی سیستم‌های چندپردازنده‌ای اجرا شود، ممکن است چندین هسته همزمان به این جدول دسترسی داشته باشند. اگر این اتفاق بیافتد, با حذف شدن یک پردازه یا اضافه شدن پردازه ای جدید امکان دارد که در داده های جدول اختلال به وجود بیاید.از طرفی دسترسی همزمان داشتن چند پردازه به ptable موجب رقابت بین پردازه ها برای دسترسی به اطلاعات جدول می شود.در نتیجه می توان با قفل کردن جدول در ابتدای هر حلقه تابع از این رقابت جلوگیری کرد.از طرفی هنگام انتخاب پردازه برای اجرا، وضعیت پردازه‌ها (state) ممکن است تغییر کند. قفل کردن جدول تضمین می‌کند که این تغییرات به درستی ثبت شوند و پردازه مناسب برای اجرا انتخاب شود.

اما در سیسستم های تک پردازنده ای که فقط یک هسته داریم, در هیچ بازه ای بیشتر از یک هسته در حال اجرا نیست.پس در نتیجه هیچگاه رقابتی برای دسترسی به جدول پردازه‌ها وجود ندارد و اصلا ضروری نیست که ازقفل کردن جدول استفاده کنیم. پس می توان در سیستم های تک پردازنده با حذف قفل ها سیستم عاملی سریع تر داشت که ساختاری ساده تر دارد.

پرسش 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 (proc:335) در مکانیزم زمان بندی xv6 نسبت به موقعیت پردازه در صف در چه iteration ای امکان schedule پیدا میکند؟ در همان iteration یا درiteration بعدی)

مکانیزم زمانبندی به این صورت است که برای هر پردازه درptable , زمانبند بررسی می کند که آیا وضعیت پردازه موردنظر RUNNABLE است یا خیر.اگر بود, به هسته اختصاص داده شده و به حالت RUNNING تغییر وضعیت می‌دهد. اما حالتی را هم داریم که پردازه‌ای در هنگام بررسی جدول به حالت RUNNABLE برود. در این صورت, اگر پردازه قبل از موقعیت جاری زمان‌بند در جدول باشد (مثلاً زمان‌بند از موقعیت پردازه عبور کرده باشد)، این پردازه تا پیمایش بعدی درحلقه بررسی نمی‌شود. اما اگر پردازه بعد از موقعیت جاری زمان‌بند در جدول باشد، امکان زمان‌بندی در همان پیمایش وجود دارد، زیرا زمان‌بند هنوز به آن نرسیده است.

پرسش 7: رجیسترهای موجود در ساختار context را نام ببرید.

در ساختار context رجیسترهای edi , esi, ebx, ebp, eip را داریم.

پرسش 8:همانطور که می دانید یکی از مهمترین رجیسترها قبل از هر تعویض متن Program Counter است که نشان می دهد روند اجرای برنامه تا کجا پیش رفته .است با ذخیره سازی این رجیستر میتوان محل ادامه برنامه را بازیابی کرد. این رجیستر در ساختار context چه نام دارد؟ این رجیستر چگونه قبل از انجام تعویض متن ذخیره میشود؟

در ساختار xv6, رجیستر eip شامل آدرس دستورالعمل بعدی ای است که باید اجرا شود و مشخص می کند که در کدام نقطه از برنامه cpu باید به اجرای دستورات بپردازد.برای انجام context switch اول رجیسترهایی که مهم هستند مقادیرشان ذخیره میشود, سپس به استک انتقال داده شده و در نهایت در ساختار context, کپی خواهند شد.

پرسش 9:همانطور که در قسمت قبل مشاهده کردید، ابتدای تابع scheduler ایجاد وقفه به کمک تابع sti، فعال می شود. با توجه به توضیحات این قسمت اگر وقفه ها فعال نمی شد چه مشکلی به وجود می آمد؟

اگر وقفه‌ها در ابتدای تابع scheduler با فراخوانی تابع sti فعال نمی‌شدند، مشکلات زیر به وجود می‌آمد:

1- یکی ازمهم‌ترین وقفه‌ها درxv6 تایمراست که باعث می‌شود که تابع yield برای پردازه در حال اجرا فراخوانی شده و پردازه زمان خود را به پایان برساند. حال اگر وقفه ها فعال نشوند, تایمر درست کار نمیکند و در نتیجه پردازه فعلی ممکن است بدون محدودیت زمانی روی CPU باقی بماند. در نتیجه این امر هم پردازه‌های دیگر به CPU دسترسی نمی توانند داشته باشند و در نتیجه دچار starvation می شوند.

2- علاوه بر تایمر، وقفه‌های دیگری مانندI/O ، وقفه‌های سخت‌افزاری و نرم‌افزاری ممکن است رخ دهند. در صورت فعال نشدن وقفه ها سیستم نمی‌تواند به این وقفه‌ها پاسخ دهد، که ممکن است منجر به از کار افتادن دستگاه‌های input/output یا رخدادهای مهم شود.

3- در هنگام وقوع وقفه تایمر، زمان سیستم (system time) به روز می‌شود. این زمان برای عملکرد بسیاری از بخش‌های سیستم‌عامل (مانند زمان‌بندی و رخدادهای زمان‌محور) حیاتی است. اگر وقفه ها غیرفعال باشند, زمان سیستم ثابت می‌ماند و تغییری نخواهد کرد. در نتیجه این اختلال, رخدادهایی که به زمان وابسته‌اند (مانند sleep برای پردازه ها که رخدادی وابسته به زمان است)به درستی عمل نمی‌کنند.

پرسش 10: بنظر شما وقفهٔ تایمر هر چه مدت یک بار صادر میشود؟ (راهنمایی: می توانید با اضافه کردن یک cprintf پس از ticks + + این موضوع را مشاهده کنید.)

وقفهٔ تایمر در xv6 به طور پیش‌فرض هر 10میلی‌ثانیه یک بار صادر می‌شود. این مدت‌زمان در معماری RISC-V توسط تنظیمات سیستم تایمر (Timer) مشخص می‌شود و برای مدیریت زمان‌بندی و به‌روزرسانی متغیرهای مرتبط با زمان (مانند ticks) استفاده می‌شود.

پرسش 11: با توجه به توضیحات داده شده چه تابعی منجر به انجام شدن گذار interrupt در شکل 1 خواهد شد؟

گذار Interrupt (در چرخه وضعیت پردازه‌ها طبق شکل 1) از حالت RUNNING به حالت READY صورت می گیرد. در xv6, این گذار توسط تابع yield مدیریت می‌شود. این تابع به این صورت عمل می کند که پردازه جاری را مجبور می‌کند که CPU را رها کرده و در صف پردازه‌های آماده (RUNNABLE) قرار بگیرد.

پرسش 12: با توجه به توضیحات قسمت scheduler dispatch میدانیم زمان بندی در xv6 به شکل نوبت گردشی است. حال با توجه مشاهدات خود در این قسمت استدلال کنید کوانتوم زمانی این پیاده سازی از زمانبندی نوبت گردشی چند میلی ثانیه است؟

در جدول 1 در صورت پروژه, الگوریتم های زمان بندی استفاده شده در سیستم عامل های مختلف را داشتیم که بخشی که مربوط به سیستم عامل xv6 بود به این صورت بود:

xv6	RR	همه فرآیندها از اولویت یکسانی برخوردارند و به ترتیب نوبتی اجرا می‌شوند، که باعث می‌شود زمان‌بندی مناسب برای سیستم‌های ساده و بی‌درنگ باشد، اما در مدیریت پیچیده‌تر اولویت‌ها و تسک‌ها محدودیت دارد.

همانطور که میبینیم, در xv6 از الگوریتم Round-Robin برای زمانبندی استفاده شده است و این یعنی هر پردازه در هنگام اجرا، ماکسیمم تا اولین وقفه تایمر زمان دارد که از CPU استفاده کند. پس می توان گفت که مدت‌زمان کوانتوم زمانی مستقیماً برابر با مدت‌زمان بین دو وقفه تایمر است. در پرسش 10 دیدیم که وقفه تایمر در xv6 حدودا هر 10میلی‌ثانیه یک بار صادر می‌شود. پس نتیجه میگیریم که در سیستم عامل xv6 ، کوانتوم زمانی زمان‌بندی Round-Robin به اندازه 10میلی‌ثانیه است.

پرسش 13: تابع wait در نهایت از چه تابعی برای منتظر ماندن برای اتمام کار یک پردازه استفاده میکند؟

درxv6 ، تابع wait از sleep برای منتظر ماندن والد تا اتمام کار فرزند استفاده می‌کند. اگر هیچ فرزندی وجود نداشته باشد یا همه آنها قبلاً تمام شده باشند، تابع wait فوراً باز می‌گردد.

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

تابع sleep به غیر از منتظر ماندن برای اتمام کار یک پردازه, در حالتی که نیاز به دسترسی به منابعی داریم که هنوز آزاد نشده اند و باید منتظر بمانیم تا به طور کامل آزاد شوند هم کاربرد دارد.

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

درxv6 ، تابع wakeup وظیفه آگاه سازی پردازه‌هایی که منتظر وقوع یک رویداد خاص بوده‌اند را بر عهده دارد. این تابع، پردازه‌هایی که به حالت SLEEPING رفته‌اند و منتظر یک آدرس خاص هستند را از خواب بیدار کرده و به حالت RUNNABLE منتقل می‌کند.

پرسش 16: با توجه به پاسخ سوال۹ این تابع منجر به گذار از چه وضعیتی به چه وضعیتی در شکل ۱ خواهد شد؟

فرض کنید پردازه ای در حال اجراست(وضعیت RUNNING در شکل) و از CPU استفاده می کند. اگر حین اجرا پردازه نیاز پیدا کند که برای رخ دادن یک رویداد خاص مانند رسیدن زمانی مشخص یا اتمام فرآیندI/O منتظر بماند، تابع sleep فراخوانی می‌شود. با فراخوانی تابع sleep , وضعیت پردازه از RUNNING به WAITING تغییر می‌کند. در نتیجه این تغییر, پردازه به حالت خواب می رود تا هنگامی که رویداد مورد انتظارش رخ بدهد.

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

علاوه بر sleep ، توابع دیگری مانند wait و توابع مرتبط با pipe نیز می‌توانند باعث گذار پردازه از وضعیت RUNNING به وضعیت WAITING شوند.

پرسش 18: در بخش ۳.۳.۲ منبع درس با پردازه های Orphan آشنا شدید رویکرد xv6 در رابطه با این گونه پردازه ها چیست؟

در سیستم عامل xv6, پردازه ای داریم به نام init. این پردازه اولین پردازه ای است که توسط کرنل xv6 در فضای کاربر اجرا می‌شود. این پردازه به طور پیش‌فرض به عنوان والد تمامی پردازه‌های orphan که والد خود را از دست داده‌اند، عمل می‌کند.پس وقتی پردازه‌ای orphan می‌شود (والدش خاتمه می‌یابد)، xv6 والد آن پردازه را به پردازه init تغییر می‌دهد. پردازه init هم به صورت مداوم تابع wait را اجرا می‌کند تا از خاتمه پردازه‌های یتیم باخبر شود و منابع آنها را آزاد کند.

پرسش 19: مقدار CPUS را مجددا به عدد ۲برگردانید. آیا همچنان ترتیبی که قبلا مشاهده میکردید پا برجاست؟ علت این امر چیست؟

با تغییر مقدار CPUS به 2 ,ترتیبی که قبلا مشاهده میکردیم دچار تغییر می شود. علت این تغییر این است که در سیستم عامل xv6 هر کدام از هسته ها, اجرای مستقلی دارند و به صورت جداگانه به پردازه‌ها نوبت می‌دهد. در نتیجه، ترتیب اجرا بین دو هسته هم زمان ممکن است با ترتیبی که در حالت تک‌هسته ای مشاهده می‌شد، متفاوت باشد. از طرفی همانطور که در سوال های قبلی به طرز کارکرد

قفل ها در xv6 پرداختیم, دیدیم که قفل‌ها ممکن است باعث شوند که یکی از هسته‌ها برای مدت کوتاهی منتظر بماند تا قفل آزاد شود. این امر سبب می شود که ترتیب اجرا تحت تأثیر قرار گیرد. همچنین همانطور که درمنبع درس خوانده ایم, در سیستم های چندهسته ای، پردازه‌ای که روی یک هسته در حال اجرا است ممکن است به دلیل Interrupt یا نیاز به منابعی خاص، به هسته دیگری منتقل شود. این تغییر نیز می‌تواند ترتیب اجرای مشاهده شده را تغییر دهد.

پرسش 20: در صورت نیاز به مقداردهی اولیه به فیلدهای اضافه شده در ساختار cpu در چه تابعی از xv6 بهتر است این کار انجام گیرد؟ راهنمایی: به main.c مراجعه کنید.

برای مقداردهی اولیه به فیلدهای اضافه‌شده به ساختار cpu، بهترین مکان تابع cpuinit در فایل main.c است. این تابع مسئول مقداردهی اولیه مربوط به ساختار CPU در مراحل ابتدایی راه‌اندازی سیستم است. به خصوص اگر اگر فیلدهای اضافه‌شده به ساختار cpu نیازمند مقداردهی ثابت و اولیه (مثلاً یک مقدار پیش‌فرض) هستند، بهتر است این کار در cpuinit انجام شود.

پرسش 21: با توجه به سیاستهای پیاده سازی شده در سطحهای دوم و سوم و همچنین استفاده از روش time-slicing توجیه کنید چرا همچنان مشکل starvation امکان رخ دادن دارد؟

الگوریتمی که در سطح دوم به کار گرفته شد الگوریتم sjf بود که همانطور که در فصل 5 منبع درس دیدیم, در این الگوریتم, اولویت با پردازه هایی است که burst time کوتاه تری دارند. اشکالی که این الگوریتم ممکن بود ایجاد کند این بود که اگر پردازه ای با burst time بزرگ به همراه یک سری پردازه با burst time کوچک در صف داشته باشیم, هر باری که scheduler می خواهد پردازه ای را برای اجرا انتخاب کند, پردازه های با burst time کوچکتر را انتخاب می کند و پردازه ای که burst time بزرگی دارد ممکن است هیچ گاه انتخاب نشده و در نتیجه به cpu دسترسی پیدا نکند و دچار starvation بشود.

از طرفی الگوریتمی که در سطح سوم داشتیم, الگوریتم FCFS بود که در آن اولویت پردازه ای بود که زودتر وارد صف شده بود. اشکالی که این الگوریتم ممکن است با آن مواجه شود این است که اگر پردازه ای داشته باشیم که مثلا دارای یک لوپ بی نهایت باشد و در طول اجرا interrupt ای ایجاد نکند, آنگاه این پردازه برای مدت نامحدود cpu را میگیرد و احازه دسترسی به cpu را به پردازه های دیگر نمی دهد چون در این الگوریتم تا زمانی که پردازه مورد نظر کارش تمام نشود, منابع و cpu را آزاد نمی کند.پس در نتیجه, پردازه های دیگر دچار starvation خواهند شد.

پرسش 22: به چه علت مدت زمانی که پردازه در وضعیت SLEEPING میباشد به عنوان زمان انتظار پردازه از منظر زمان بندی در نظر گرفته نمیشود؟

به طور کلی waiting time یا همان زمان انتظار پردازه, نشان دهنده مدت زمانی است که پردازه در انتظار اجرا شدن توسط cpu است. اما پردازه ای که در وضعیت sleeping می باشد, وضعیتش ناشی از انتظار برای تکمیل فرآیندی I/O یا ازاد شدن منابعی مانند دیسک است, نه انتظار برای cpu. از طرفی زمان‌بند وظیفه مدیریت استفاده از CPU را دارد و به منابع دیگر توجهی ندارد. به همین دلیل است که زمانبند تنها با پردازه هایی که در وضعیت ready (در xv6 همان runnable) هستند تعامل دارند و تعاملی با پردازه های دارای وضعیت sleeping ندارد. پس مدت زمانی که پردازه در وضعیت SLEEPING میباشد به عنوان زمان انتظار پردازه از منظر زمان بندی در نظر گرفته نمیشود.

شرح پروژه:

در ساختار process فیلدی تحت عنوان schedule information تعریف می کنیم. این ساختار دارای صف فعلی , burst time, آخرین زمان اجرا, زمان ساخته شدن, زمان ورود به صف فعلی, میزان اطمینان و تعداد سیکل های اجرا شده متوالی می باشد.

```
struct proc {
  int syscalls[MAX_SYSCALLS];
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[24];               // Process name (debugging)

  struct ScheduleInfo sched_info;


};

struct ScheduleInfo
{
  enum schedqueue level;
  int burst_time;
  int last_exe_time;
  uint creation_time;
  uint enter_level_time;
  int confidence;
  int num_of_cycles;
};
```

در ساختار cpu, سه متغیر rr – sjf – fcfs اضافه می کنیم که هر کدام نشان دهنده تایم اسلایس باقی مانده برای هر صف می باشد. مقادیر این متغیر ها, در فایل mp.c و هنگام initial کردن cpu به ترتیب 30,20,10 ذخیره می شوند.(به علت آن که هر تیک 10ms است, ما با هر تیک مقدار متغیر موردنظر را تغییر می دهیم)

```
struct cpu {
  uchar apicid;
  struct context *scheduler;
  struct taskstate ts;
  struct segdesc gdt[NSEGS];
  volatile uint started;
  int ncli;
  int intena;
  struct proc *proc;
  int rr;
  int sjf;
  int fcfs;
};
```

تابع scheduler را به شکلی تغییر می دهیم که اگر time sliceای برای اجرای process از صف اول وجود داشت, با استفاده از تابع round-robin ,process مطلوب را انتخاب می کند.اگر این شرط برقرار نبود یا در صف اول process قابل اجرایی وجود نداشت, به سراغ صف دوم می رویم و در صورتی که برای این صف,time slice باقی مانده بود,با استفاده از short_job_first , process مطلوب را انتخاب می کنیم و در انتها اگر این شرط برای این صف هم برقرار نبود, با استفاده از تابع first_come_first_service و با در نظر گرفتن تایم اسلایس این صف , process مطلوب را از این صف انتخاب می کنیم.در نهایت اگر هیچ process قابل اجرایی وجود نداشت, تایم اسلایس های هر سه صف را reset می کنیم.

```
void scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  struct proc *last_scheduled_RR = &ptable.proc[NPROC - 1];
  c->proc = 0;
  for (;;)
  {
    sti();
    p = 0;
    acquire(&ptable.lock);
    if(mycpu()->rr>0)
      p = round_robin(last_scheduled_RR);
    if (p)
      last_scheduled_RR = p;
    else
    {
      if(mycpu()->sjf>0)
        p = short_job_first();
      if (!p)
      {
        if(mycpu()->fcfs>0)
          p = first_come_first_service();
        if (!p)
        {
          mycpu()->rr = 30;
          mycpu()->sjf = 20;
          mycpu()->fcfs = 10;
          release(&ptable.lock);
          continue;
        }
      }
    }
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    p->sched_info.last_exe_time = ticks;
    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc = 0;
  release(&ptable.lock);
  }
}
```

در تابع round_robin با گرفتن آخرین process اجرا شده به عنوان آرگومان از process بعدی آن پیمایش را شروع کرده تا به اولین process قابل اجرا برسیم که همان خاصیت round_robin است.
توجه شود که اگر در پیمایش به process آخر رسیدیم, به اولین process در table برمیگردیم که خاصیت دایره ای بودن صف می باشد.

```
struct proc *
round_robin(struct proc *last_scheduled)
{
  struct proc *p = last_scheduled;
  for (;;)
  {
    p++;
    if (p >= &ptable.proc[NPROC])
      p = ptable.proc;
    if (p->state == RUNNABLE && p->sched_info.level == ROUND_ROBIN)
      return p;

    if (p == last_scheduled)
      return 0;
  }
  return 0;
}
```

برای صف sjf, در میان تمام process ها پیمایش می کنیم و برای هر process که قابل اجرا بود, یک عدد رندوم تولید کرده و با میزان اطمینان مقایسه می کنیم و اگر شرایط برقرار بود, چک می کنیم آیا burst time اجرای این process مینیمم هست یا خیر. در انتها process ای که دارای کمترین burst time بوده و شرط confidence آن برقرار بوده را return می کنیم.

```
struct proc* short_job_first()
{
  struct proc* res=0;
  struct proc* p;
  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++)
  {
    if((p->state != RUNNABLE) || (p->sched_info.level!=SJF))
      continue;
    if(res == 0)
      res = p;
    if(p->sched_info.confidence > create_rand_num(100))
    {
      if(p->sched_info.burst_time < res->sched_info.burst_time)
        res = p;
    }
  }
  return res;
}
```

برای صف FCFS در میان process ها پیمایش کرده و آنی را که زودتر از همه به این صف آمده و قابل اجرا هست را return می کنیم.

```
struct proc* first_come_first_service()
{
  struct proc* res=0;
  struct proc* p;
  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++)
  {
    if((p->state != RUNNABLE) || (p->sched_info.level!=FCFS))
      continue;
    if(res == 0)
      res = p;
    else if(p->sched_info.enter_level_time<res->sched_info.enter_level_time)
      res = p;
  }
  return res;
}
```

برای تغییر صف یک process تابع set_level را تعریف کرده ایم (سیستم کالی نیز با همین عنوان تعریف شده که از همین تابع استفاده می کند).در این تابع با گرفتن pid و صف مقصد به عنوان آرگومان صف process موردنظر را تغییر می دهیم.

```
int set_level(int pid, int target_level)
{
  struct proc *p = findproc(pid);
  acquire(&ptable.lock);

  int old_queue = p->sched_info.level;
  p->sched_info.level = target_level;
  p->sched_info.enter_level_time = ticks;

  release(&ptable.lock);
  return old_queue;
}
```

جهت جلوگیری از starvation, تابع aging تعریف شده است. در این تابع تمام process هایی که قابل اجرا هستند, در صف اول نیستند و از زمانی که وارد این صف شده اند به آستانه starvation رسیده اند, آن ها را به یک صف بالاتر می آوریم.

```
void aging()
{
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == RUNNABLE)
    {
      if( p->sched_info.level == ROUND_ROBIN)
        continue;
      if ((ticks - p->sched_info.last_exe_time > STARVATION_THRESHOLD) &&
        | (ticks - p->sched_info.enter_level_time > STARVATION_THRESHOLD))
      {
        {
          release(&ptable.lock);
          if(p->sched_info.level == SJF)
            set_level(p->pid, ROUND_ROBIN);
          else if(p->sched_info.level == FCFS)
            set_level(p->pid, SJF);
          cprintf("pid: %d starved!\n", p->pid);
          acquire(&ptable.lock);
        }
      }

    }
  }
  release(&ptable.lock);
}
```

توجه: در هنگام initial کردن متغیرهای یک process در تابع alloc_proc صف برابر FCFS قرار داده می شود و مقادیر 2,50 در اینجا برای burst time و confidence قرار داده می شوند.
Process های init و sh با استفاده از تابع set_level در همان زمان ساخته شدن به صف اول می روند.

از آنجا که هر تیک زمانی سیستم 10ms است و ما میخواهیم آن را به 50ms تغییر دهیم, در هر process تعداد سیکل های متوالی که این process انجام داده است را ذخیره می کنیم و هر موقع به 5 سیکل رسید, آنگاه cpu process را آزاد می کند. همچنین در هر تیک زمانی با توجه به صف process ای که در cpu قرار دارد, times slice صف فعلی را کم می کنیم.

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
{
  if(myproc()->sched_info.level == ROUND_ROBIN)
    mycpu()->rr--;
  else if(myproc()->sched_info.level == SJF)
    mycpu()->sjf--;
  else if(myproc()->sched_info.level == FCFS)
    mycpu()->fcfs--;

  if(myproc()->sched_info.num_of_cycles<5)
    myproc()->sched_info.num_of_cycles++;
  else
  {
    cprintf("pid: ");
    cprintf("%d", myproc()->pid);
    cprintf(" ticks: ");
    cprintf("%d",ticks);
    cprintf(" level: ");
    cprintf("%d",myproc()->sched_info.level);
    cprintf("\n");
    myproc()->sched_info.num_of_cycles = 0;
    yield();
  }
}
```

فراخوانی های سیستمی:

1-set_level: در بالا توضیح داده شد.

2- set_burst_confidence: با استفاده از این فراخوانی سیستمی با گرفتن pid به عنوان آرگومان burst time و confidence پروسه را آپدیت می کنیم.

```
void set_burst_confidence(int pid, int burst, int conf)
{
  struct proc *p = findproc(pid);
  p->sched_info.burst_time = burst;
  p->sched_info.confidence = conf;
  cprintf("pid: %d new_burst: %d new_confidence: %d\n", pid, p->sched_info.burst_time, p->sched_info.confidence);
  return 0;
}
```

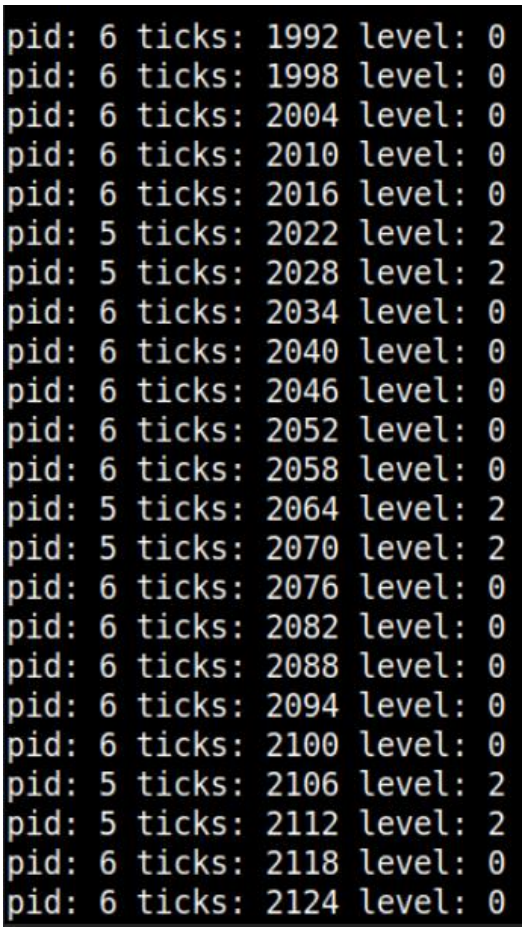
3- show_process_info: با استفاده از این فراخوانی سیستمی, اطلاعات مربوط به process را نمایش می دهیم.

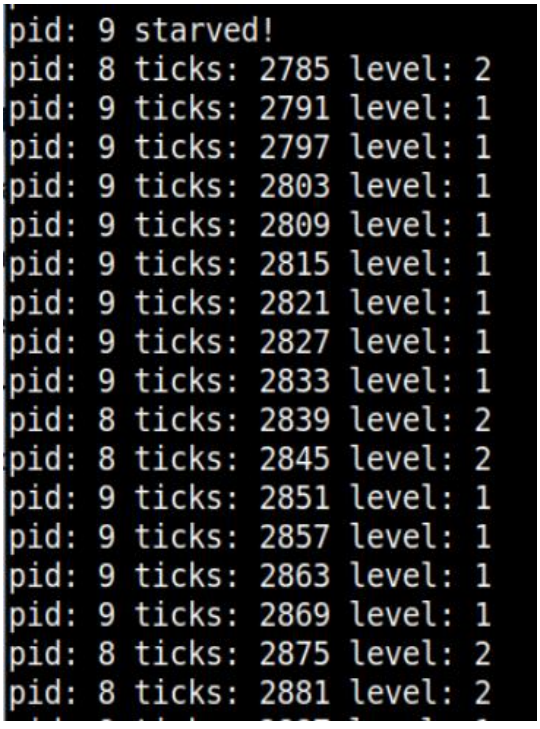
توجه: برای waiting , مقدار clock کنونی سیستم منهای زمان آخرین اجرای process را نمایش می دهیم. بقیه اطلاعات در process ذخیره شده اند.

```
void show_process_info()
{

  static char *states[] = {
      [UNUSED] "unused",
      [EMBRYO] "embryo",
      [SLEEPING] "sleeping",
      [RUNNABLE] "runnable",
      [RUNNING] "running",
      [ZOMBIE] "zombie"};

  static int columns[] = {24, 10, 10, 10, 10, 10, 15, 12, 12};
  cprintf("Process_Name           PID     State    Queue   Burst_time   waiting   Enterance_time   confidence    consecutive_run\n"
          "-------------------------------------------------------------------------------------\n");

  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == UNUSED)
      continue;

    const char *state;
    if (p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "unknown state";

    cprintf("%s", p->name);
    space(columns[0] - strlen(p->name));

    cprintf("%d", p->pid);
    space(columns[1] - num_digits(p->pid));

    cprintf("%s", state);
    space(columns[2] - strlen(state));

    cprintf("%d", p->sched_info.level);
    space(columns[3] - num_digits(p->sched_info.level));

    cprintf("%d", (int)p->sched_info.burst_time);
    space(columns[4] - num_digits((int)p->sched_info.burst_time));

    cprintf("%d", ticks - p->sched_info.last_exe_time);
    space(columns[5] - num_digits(ticks - p->sched_info.last_exe_time));

    cprintf("%d", p->sched_info.enter_level_time);
    space(columns[6] - num_digits(p->sched_info.enter_level_time));

    cprintf("%d", (int)p->sched_info.confidence);
    space(columns[7] - num_digits((int)p->sched_info.confidence));

    cprintf("%d", (int)p->sched_info.num_of_cycles);
    space(columns[7] - num_digits((int)p->sched_info.num_of_cycles));
    cprintf("\n");
  }
}
```

بررسی عملکرد مدل:

```
int main()
{
    for (int i = 0; i < PROCS_NUM; ++i)
    {
        int pid = fork();
        // if(pid==6)  //check SJF
        //     set_burst_confidence(pid,1,99);
        // if(pid==7)  //check SJF note : change all p
        //     set_burst_confidence(pid,3,99);
        if(pid==6)  //check multilevel time slicing
            set_level(pid,0);

        if (pid == 0)
        {

            test();
            exit();
        }
    }
    show_process_info();
    for (int i = 0; i < PROCS_NUM; i++)
        wait();
    show_process_info();
    exit();
}
```


pid: 6 ticks: 1992 level: 0
pid: 6 ticks: 1998 level: 0
pid: 6 ticks: 2004 level: 0
pid: 6 ticks: 2010 level: 0
pid: 6 ticks: 2016 level: 0
pid: 5 ticks: 2022 level: 2
pid: 5 ticks: 2028 level: 2
pid: 6 ticks: 2034 level: 0
pid: 6 ticks: 2040 level: 0
pid: 6 ticks: 2046 level: 0
pid: 6 ticks: 2052 level: 0
pid: 6 ticks: 2058 level: 0
pid: 5 ticks: 2064 level: 2
pid: 5 ticks: 2070 level: 2
pid: 6 ticks: 2076 level: 0
pid: 6 ticks: 2082 level: 0
pid: 6 ticks: 2088 level: 0
pid: 6 ticks: 2094 level: 0
pid: 6 ticks: 2100 level: 0
pid: 5 ticks: 2106 level: 2
pid: 5 ticks: 2112 level: 2
pid: 6 ticks: 2118 level: 0
pid: 6 ticks: 2124 level: 0


pid: 9 starved!
pid: 8 ticks: 2785 level: 2
pid: 9 ticks: 2791 level: 1
pid: 9 ticks: 2797 level: 1
pid: 9 ticks: 2803 level: 1
pid: 9 ticks: 2809 level: 1
pid: 9 ticks: 2815 level: 1
pid: 9 ticks: 2821 level: 1
pid: 9 ticks: 2827 level: 1
pid: 9 ticks: 2833 level: 1
pid: 8 ticks: 2839 level: 2
pid: 8 ticks: 2845 level: 2
pid: 9 ticks: 2851 level: 1
pid: 9 ticks: 2857 level: 1
pid: 9 ticks: 2863 level: 1
pid: 9 ticks: 2869 level: 1
pid: 8 ticks: 2875 level: 2
pid: 8 ticks: 2881 level: 2

همانطور که در کد مشخص است, در ابتدا, 6 processمی سازیم و process با pid 6 را با استفاده از فراخوانی سیستمی set_level به صف round_robin می بریم. در عکس result اول, واضح است که time slice اختصاص داده شده به process 6 از دیگر process ها بیشتر است(زیرا بقیه process ها در صف FCFS هستند) همچنین به علت آن که process 5 زودتر از باقی process ها در صف FCFS قرار گرفته ,cpu در اختیار این process قرار می گیرد.

در result دوم می بینیم هنگامی که pid 9 دچار starvation شد, به صف sjf آمده و time slice هایی به این process اختصاص داده می شود(بیشتر از process های موجود در صف FCFS).

بررسی عملکرد sjf:

در ابتدا process با pid 6 را با مقادیر 99,1 برای confidence و burst time با استفاده از فراخوانی سیستمی در نظر می گیریم.process با pid 7 را با مقادیر 99,3 برای confidence و burst time با استفاده از فراخوانی سیستمی در نظر می گیریم.همچنینdefault تمام process ها را صف sjf در نظر می گیریم.

```
int main()
{
    for (int i = 0; i < PROCS_NUM; ++i)
    {
        int pid = fork();
        if(pid==6)  //check SJF
            set_burst_confidence(pid,1,99);
        if(pid==7)  //check SJF note : change all procresses confiden
            set_burst_confidence(pid,3,99);
        // if(pid==6)  //check multilevel time slicing
        //     set_level(pid,0);

        if (pid == 0)
        {

            test();
            exit();
        }
    }
    show_process_info();
    for (int i = 0; i < PROCS_NUM; i++)
        wait();
    show_process_info();
    exit();
}
```

```
Process_Name           PID     State    Queue   Burst_time   waiting   Enterance_time   confidence    consecutive_run
-------------------------------------------------------------------------------------
init                   1        sleeping  0         2         491       0              50          1
sh                     2        sleeping  0         2         2         3              50          2
show_process_info      3        running   1         2         1         492            50          1
show_process_info      4        runnable  1         2         494       493            50          0
show_process_info      5        runnable  1         2         494       493            50          0
show_process_info      6        runnable  1         1         494       493            99          0
show_process_info      7        runnable  1         3         495       493            99          0
show_process_info      8        runnable  1         2         495       494            50          0
pid: 6 ticks: 501 level: 1
pid: 6 ticks: 507 level: 1
pid: 6 ticks: 513 level: 1
pid: 6 ticks: 519 level: 1
pid: 6 ticks: 525 level: 1
pid: 6 ticks: 531 level: 1
pid: 6 ticks: 537 level: 1
pid: 6 ticks: 543 level: 1
pid: 6 ticks: 549 level: 1
pid: 6 ticks: 555 level: 1
pid: 6 ticks: 561 level: 1
pid: 6 ticks: 567 level: 1
pid: 6 ticks: 573 level: 1
```

```
pid: 7 ticks: 1523 level: 1
pid: 7 ticks: 1529 level: 1
pid: 7 ticks: 1535 level: 1
pid: 7 ticks: 1541 level: 1
pid: 7 ticks: 1547 level: 1
pid: 7 ticks: 1553 level: 1
pid: 7 ticks: 1559 level: 1
pid: 7 ticks: 1565 level: 1
pid: 7 ticks: 1571 level: 1
pid: 7 ticks: 1577 level: 1
pid: 7 ticks: 1583 level: 1
pid: 7 ticks: 1589 level: 1
Process_Name           PID     State    Queue   Burst_time   waiting   Enterance_time   confidence    consecutive_run
-------------------------------------------------------------------------------------
init                   1        sleeping  0         2         1592      0              50          1
sh                     2        sleeping  0         2         1103      3              50          2
show_process_info      3        running   1         2         1         492            50          3
```

مشاهده می شود که در ابتدا cpu به process 6 اختصاص داده می شود زیرا دارای کم ترین burst time است و چون confidence آن 99 است, قطعا اجرا می شود.همچنین چون process 7 دارای بلندترین طول است, در انتها اجرا می شود.

بررسی عملکرد round_robin:

برای این کار تمام process ها را در صف round_robinقرار می دهیم. نتیجه آن به صورت زیر خواهد بود که مشاهده می شود processor مدام میان process های مختلف جابجا می شود.

```
Process_Name          PID     State    Queue   Burst_time   waiting   Enterance_time   confidence    consecutive_run
-------------------------------------------------------------------------------------
init                  1         sleeping  0         2         643       0              50          2
sh                    2         sleeping  0         2         1         3             50          0
show_process_info     3         running   0         2         1         644           50          1
show_process_info     4         runnable  0         2         645       644           50          0
show_process_info     5         runnable  0         2         645       644           50          0
show_process_info     6         runnable  0         2         646       645           50          0
show_process_info     7         runnable  0         2         646       645           50          0
show_process_info     8         runnable  0         2         646       645           50          0
pid: 4 ticks: 652 level: 0
pid: 5 ticks: 658 level: 0
pid: 6 ticks: 664 level: 0
pid: 7 ticks: 670 level: 0
pid: 8 ticks: 676 level: 0
pid: 4 ticks: 682 level: 0
pid: 5 ticks: 688 level: 0
pid: 6 ticks: 694 level: 0
pid: 7 ticks: 700 level: 0
pid: 8 ticks: 706 level: 0
pid: 4 ticks: 712 level: 0
pid: 5 ticks: 718 level: 0
pid: 6 ticks: 724 level: 0
pid: 7 ticks: 730 level: 0
pid: 8 ticks: 736 level: 0
pid: 4 ticks: 742 level: 0
pid: 5 ticks: 748 level: 0
pid: 6 ticks: 754 level: 0
pid: 7 ticks: 760 level: 0
```

عملکرد FCFS هم در multi_level بررسی شد.