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 :

```
// Per-process state
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)
};
```

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:

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

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:

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}
```

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 وضعیت 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 وضعیت به runnable معادل ready تغییر می کند.

-4

در فایل param.h داریم:

```
#define NPROC        64  // maximum number of processes
#define KSTACKSIZE 4096  // size of per-process kernel stack
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
#define ROOTDEV       1  // device number of file system root disk
#define MAXARG       32  // max exec arguments
#define MAXOPBLOCKS  10  // max # of blocks any FS op writes
#define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
#define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
#define FSSIZE       1000  // size of file system in blocks
```

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

اگر به کدallproc 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

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
```

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

-7

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

-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.

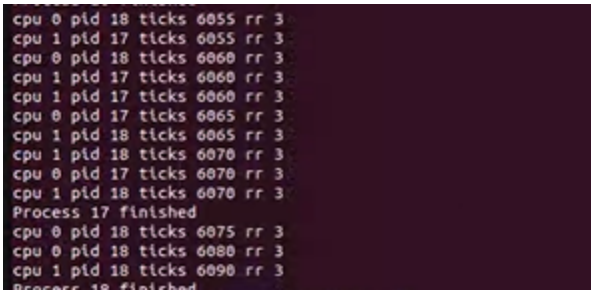
-13

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.

```
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```

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 .

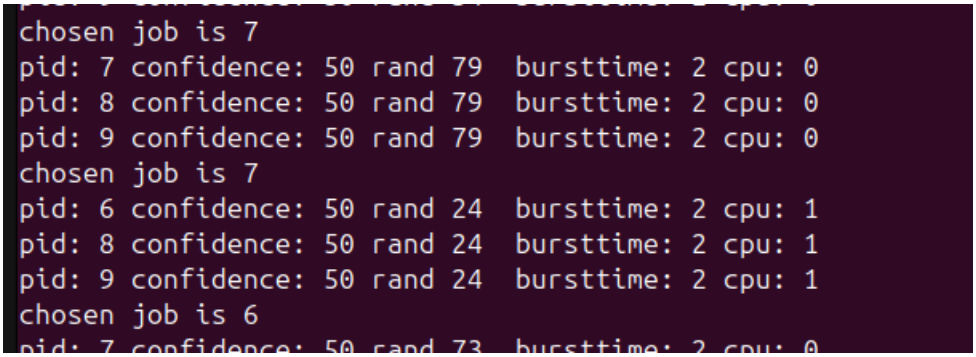
تست صف ها و تست کلی برنامه:

Sjf:

روی یک cpu:

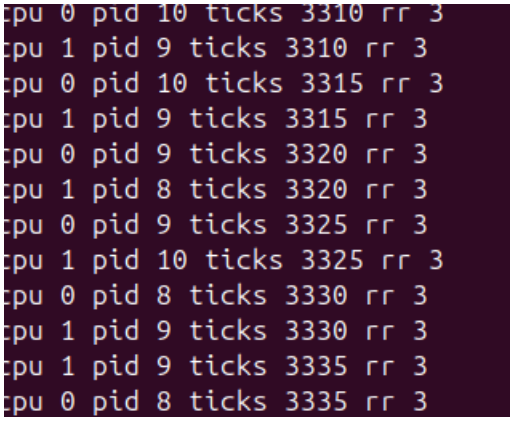
همانطور که در این بخش می بینید با توجه به آنکه عدد رندوم از confidence همه ی پراسس ها کوچکتر است همه ی پراسس ها ولید هستند و کوچک ترین bursttime که برابر 2 است را انتخاب می کنیم.

روی هر دو cpu :



Roundrobin:

با 2 cpu:



با تک cpu :

```
cpu 0 pid 8 ticks 5585 rr 3
cpu 0 pid 6 ticks 5590 rr 3
cpu 0 pid 7 ticks 5595 rr 3
cpu 0 pid 9 ticks 5600 rr 3
cpu 0 pid 7 ticks 5605 rr 3
cpu 0 pid 9 ticks 5610 rr 3
cpu 0 pid 7 ticks 5615 rr 3
cpu 0 pid 9 ticks 5620 rr 3
cpu 0 pid 7 ticks 5625 rr 3
cpu 0 pid 9 ticks 5630 rr 3
cpu 0 pid 7 ticks 5635 rr 3
cpu 0 pid 9 ticks 5640 rr 3
cpu 0 pid 7 ticks 5645 rr 3
cpu 0 pid 9 ticks 5650 rr 3
cpu 0 pid 7 ticks 5655 rr 3
cpu 0 pid 9 ticks 5660 rr 3
```

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

FCFS:

روی دو هسته:

```
pid: 4 arrival:  493 cpu: 0
pid: 5 arrival:  503 cpu: 0
chosen one is 4
pid: 3 arrival:  375 cpu: 1
pid: 5 arrival:  503 cpu: 1
chosen one is 3
pid: 3 arrival:  375 cpu: 1
pid: 5 arrival:  503 cpu: 1
```

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 :

```
cpu 0 pid 4 ticks 1647 q 3
cpu 0 pid 4 ticks 1648 q 3
cpu 0 pid 4 ticks 1649 q 3
cpu 0 pid 4 ticks 1650 q 3
cpu 0 pid 4 ticks 1651 q 3
FCFS time is finished we should go to RR timePassed : 10
cpu 0 pid 4 ticks 1652 q 3
cpu 0 pid 4 ticks 1653 q 3
cpu 0 pid 4 ticks 1654 q 3
cpu 0 pid 4 ticks 1655 q 3
cpu 0 pid 4 ticks 1656 q 3
cpu 0 pid 4 ticks 1657 q 3
cpu 0 pid 4 ticks 1658 q 3
cpu 0 pid 4 ticks 1659 q 3
cpu 0 pid 4 ticks 1660 q 3
pid 5 aged by one and age = 800 and go to SJF Queue with 2 cpu : 0 doneTicks : 0
pid 6 aged by one and age = 800 and go to SJF Queue with 2 cpu : 0 doneTicks : 0
pid 7 aged by one and age = 800 and go to SJF Queue with 2 cpu : 0 doneTicks : 0
FCFS time is finished we should go to RR timePassed : 10
Process 5 starting, duration: cpu 0 pid 5 ticks 1663 q 2
400 ticks
1663cpu 0 pid 5 ticks 1664 q 2
cpu 0 pid 5 ticks 1665 q 2
cpu 0 pid 5 ticks 1666 q 2
cpu 0 pid 5 ticks 1667 q 2
cpu 0 pid 5 ticks 1668 q 2
cpu 0 pid 5 ticks 1669 q 2
cpu 0 pid 5 ticks 1670 q 2
cpu 0 pid 5 ticks 1671 q 2
cpu 0 pid 5 ticks 1672 q 2
cpu 0 pid 5 ticks 1673 q 2
cpu 0 pid 5 ticks 1674 q 2
```

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 :

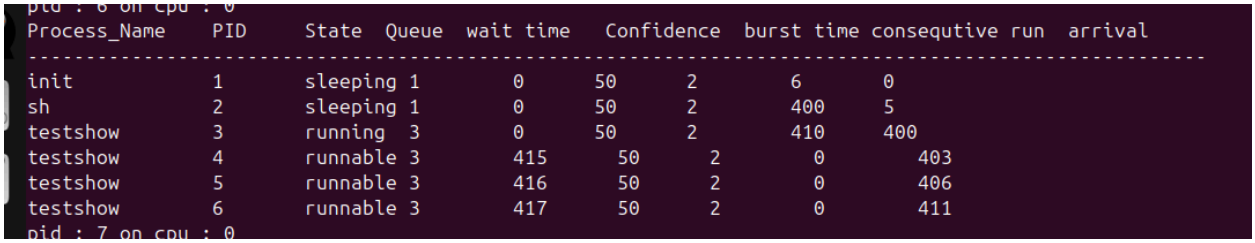
```
FCFS time is finished we should go to RR timePassed : 10
Process 5 starting, duration: cpu 0 pid 5 ticks 1663 q 2
400 ticks
1663cpu 0 pid 5 ticks 1664 q 2
cpu 0 pid 5 ticks 1665 q 2
cpu 0 pid 5 ticks 1666 q 2
cpu 0 pid 5 ticks 1667 q 2
cpu 0 pid 5 ticks 1668 q 2
cpu 0 pid 5 ticks 1669 q 2
cpu 0 pid 5 ticks 1670 q 2
cpu 0 pid 5 ticks 1671 q 2
cpu 0 pid 5 ticks 1672 q 2
cpu 0 pid 5 ticks 1673 q 2
cpu 0 pid 5 ticks 1674 q 2
cpu 0 pid 5 ticks 1675 q 2
cpu 0 pid 5 ticks 1676 q 2
cpu 0 pid 5 ticks 1677 q 2
cpu 0 pid 5 ticks 1678 q 2
cpu 0 pid 5 ticks 1679 q 2
cpu 0 pid 5 ticks 1680 q 2
SJF time is finished we should go to FCFS timePassed : 20
cpu 0 pid 4 ticks 1681 q 3
cpu 0 pid 4 ticks 1682 q 3
cpu 0 pid 4 ticks 1683 q 3
cpu 0 pid 4 ticks 1684 q 3
cpu 0 pid 4 ticks 1685 q 3
cpu 0 pid 4 ticks 1686 q 3
cpu 0 pid 4 ticks 1687 q 3
cpu 0 pid 4 ticks 1688 q 3
cpu 0 pid 4 ticks 1689 q 3
FCFS time is finished we should go to RR timePassed : 10
cpu 0 pid 5 ticks 1690 q 2
cpu 0 pid 5 ticks 1691 q 2
cpu 0 pid 5 ticks 1692 q 2
cpu 0 pid 5 ticks 1693 q 2
cpu 0 pid 5 ticks 1694 q 2
cpu 0 pid 5 ticks 1695 q 2
cpu 0 pid 5 ticks 1696 q 2
cpu 0 pid 5 ticks 1697 q 2
```

تست time slicing :

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

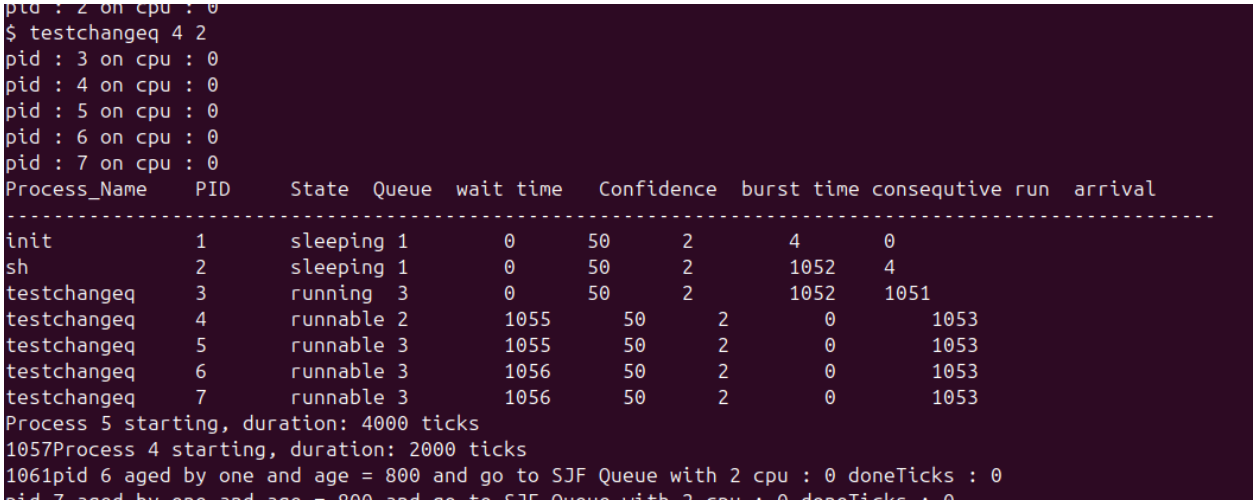
```
cpu 0 pid 2 ticks 7 q 1
$ testsch 10 100
pid : 3 on cpu : 0
cpu 0 pid 3 ticks 860 q 3
pid : 4 on cpu : 0
cpu 0 pid 3 ticks 861 q 3
pid : 5 on cpu : 0
pid : 6 on cpu : 0
pid : 7 on cpu : 0
Process 4 starting, duration: 2000 ticks
86cpu 0 pid 4 ticks 863 q 3
2cpu 0 pid 4 ticks 864 q 3
cpu 0 pid 4 ticks 865 q 3
cpu 0 pid 4 ticks 866 q 3
cpu 0 pid 4 ticks 867 q 3
cpu 0 pid 4 ticks 868 q 3
FCFS time is finished we should go to RR timePassed : 10
cpu 0 pid 4 ticks 869 q 3
cpu 0 pid 4 ticks 870 q 3
cpu 0 pid 4 ticks 871 q 3
cpu 0 pid 4 ticks 872 q 3
cpu 0 pid 4 ticks 873 q 3
cpu 0 pid 4 ticks 874 q 3
cpu 0 pid 4 ticks 875 q 3
cpu 0 pid 4 ticks 876 q 3
cpu 0 pid 4 ticks 877 q 3
FCFS time is finished we should go to RR timePassed : 10
cpu 0 pid 4 ticks 878 q 3
cpu 0 pid 4 ticks 879 q 3
cpu 0 pid 4 ticks 880 q 3
cpu 0 pid 4 ticks 881 q 3
cpu 0 pid 4 ticks 882 q 3
cpu 0 pid 4 ticks 883 q 3
cpu 0 pid 4 ticks 884 q 3
cpu 0 pid 4 ticks 885 q 3
cpu 0 pid 4 ticks 886 q 3
FCFS time is finished we should go to RR timePassed : 10
cpu 0 pid 4 ticks 887 q 3
cpu 0 pid 4 ticks 888 q 3
cpu 0 pid 4 ticks 889 q 3
cpu 0 pid 4 ticks 890 q 3
cpu 0 pid 4 ticks 891 q 3
cpu 0 pid 4 ticks 892 q 3
```

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 :

```
pid : 2 on cpu : 0
$ testb_c 33 60
pid : 3 on cpu : 0
pid : 4 on cpu : 0
4 33 60 33 60
pid : 5 on cpu : 0
5 33 60 33 60
pid : 6 on cpu : 0
6 33 60 33 60
pid : 7 on cpu : 0
7 33 60 33 60
Process_Name    PID     State  Queue  wait time   Confidence  burst time consequtive run  arrival
-----------------------------------------------------------------------------------------------
init            1       sleeping 1        0      50      2        5       0
sh              2       sleeping 1        0      50      2        967     5
testb_c         3       running  3        0      50      2        968     967
testb_c         4       runnable 3        973      60      33       0        968
testb_c         5       runnable 3        974      60      33       0        969
testb_c         6       runnable 3        975      60      33       0        969
testb_c         7       runnable 3        975      60      33       0        969
```