پروژه سوم آزمایشگاه سیستم عامل

سوال 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؟ (ذکر حداقل ۵ مورد و معادل آن ها در xv6)

در فایل proc.h یک استراکت وجود دارد که اطلاعات مربوط به هر پردازه را ذخیره می کند و هر کدام را بخش به بخش بررسی می کنیم:

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this proce
  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)
};
```

- متغیر <- sz اندازه حافظه مربوط به پردازه
- متغیر <- pgdir برای مدیریت حافظه مجازی استفاده می شود که process page directory است
- متغیر <- kstack پوینتر به استک مربوط هسته
- متغیر <- state نشان دهنده وضعیت پردازه
- متغیر <- pid شناسه پردازه
- متغیر <- parent پردازه ای که پردازه فعلی را تولید کرده و در واقع والد آن است

●متغیر <- tf پوینتر به trap frame که برای اجرا در حالت کاربر کاربرد دارد

●متغیر <- context پوینتر به رجیستر ها برای عملیات تعویض متن کاربرد دارد

●متغیر <- chan متغیری که برای sleep و wakeup در واقع آدرس جایی که wait شده یا در حالت خواب رفته را نگه می دارد

●متغیر <- killed نشان دهنده اینکه آیا پردازه برای terminate شدن نشان شده یا نه در واقع یک سیگنال است که پردازه باید تمام شود

●متغیر <- ofile نشان دهنده فایل های باز شده ای است که پردازه با آنها کار میکند

●متغیر <- cwd کوتاه شده current working directory که پوینتر به دایکتوری است که پردازه در ان در حال اجرا است

●متغیر <- name نام پردازه فعلی

در همین فایل وضعیت های پردازه ها هم به شکل یک enum تعریف شده که تک به تک بررسی می کنیم:

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

● وضعیت UNUSED برای پردازه هایی که در حال حاضر اجرا نمی شوند و استفاده نمی کنیم. وضعیت تمام پردازه ها در مقداردهی اولیه سیستم در این وضعیت هستند همچنین پردازه ها بعد از تمام شدن و آزاد سازی منابع وارد این وضعیت می شوند

●وضعیت EMBRYO نشان دهنده اینکه پردازه ساخته شده ولی مقدار دهی اولیه آن تکمیل نشده است. بعد از اینکه پردازه در allocproc رفت وارد این وضعیت می شود.

●وضعیت SLEEPING زمانی که پردازه منتظر رخ دادن یک واقعه است، در این وضعیت می رود. تا زمانی که آن اتفاق رخ نداده در این وضعیت باق می ماند.

●وضعیت RUNNABLE پردازه زمانی که آماده اجرا است و در انتظار گرفتن CPU است در این وضعیت می ماند و در صف مربوط به پردازه هایی که می توانند اجرا شوند می رود تا توسط scheduler بتواند cpu را دریافت کند.

●وضعیت RUNNING زمانی که پردازه cpu گرفته و در حال اجرا است وارد این وضعیت می شود در واقع زمان بند یک پردازه قابل اجرا را می گیرد و به آن cpu می دهد و وضعیت آن را به در حال اجرا تغییر می دهد

●وضعیت ZOMBIE زمانی که پردازه اجرایش را تمام می کند تا زمانی که پردازه والد آن منابع آن را آزاد سازی نکند در این وضعیت باق می ماند. در واقع پردازه تمام شده ولی همچنان در ptable جا اشغال کرده است و تا زمانی در این حالت می ماند که والد آن wait را صدا بزند

در منبع درس یک تصویر از PCB مربوط به پردازه ها داریم که به شکل زیر می باشد:

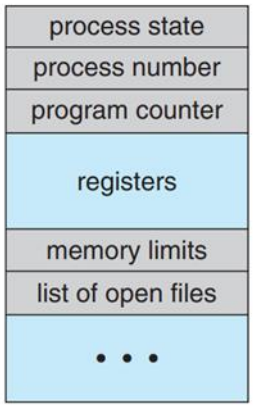


Figure 3.3 Process control block (PCB).

شباهت های زیادی بین این ساختار و ساختار موجود در xv6 وجود دارد که به ترتیب بررسی می کنیم:

1. داده process state نشان دهنده وضعیت فعلی پردازه است که همان طور که بررسی کردیم در xv6 همان state است

2. داده process number نشان دهنده شناسه یا شماره پردازه است که همان طور که بررسی کردیم در xv6با pid نمایش داده می شود

3. داده program counter نشان دهنده دستوری است که در حال حاضر در حال اجرا است که معادل داده ای در trap frame در xv6 است که با tf تعریف شده

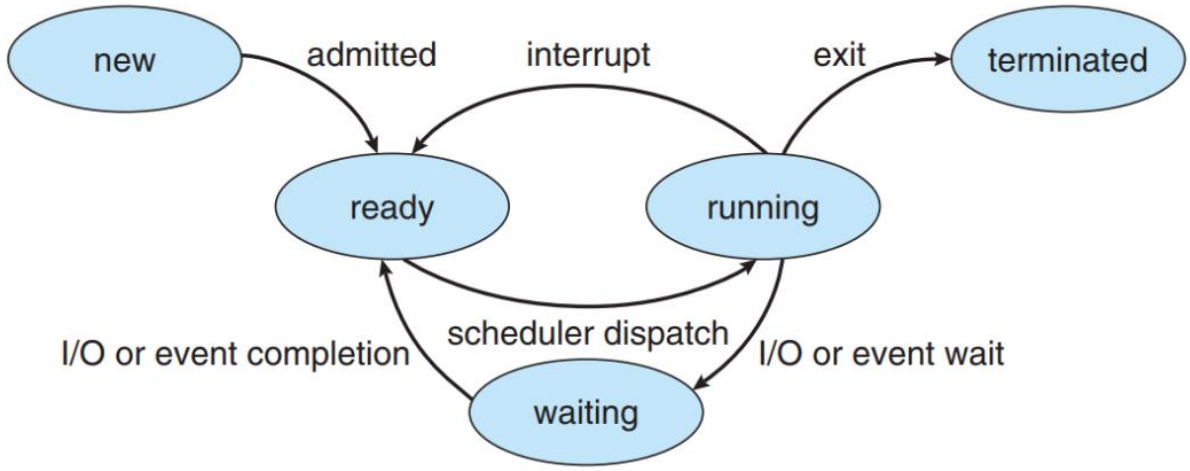
4. داده registers همان رجیستر هایی که داده ها در آن هستند و برای تعویض متن کاربرد دارند که میتوان معادل context در xv6 دانست

5. داده memory limit نشان دهنده محدودیت های حافظه است که در xv6 با متغیر sz نشان داده شده است

6. داده list of open files مربوط به فایل های باز مورد نیاز پردازه است که همان ofile در xv6 است

سوال 2-هر کدام از وضعیت های تعریف شده معادل کدام وضعیت در شکل ۱ می باشند؟

طبق منبع درس این نمودار برای وضعیت پردازه ها رسم شده است که معادل آنها را در xv6 بررسی می کنیم



●حالت new :در این حالت یک پردازه ایجاد شده اما همچنان آماده اجرا نیست. این حالت مشابه حالت EMBRYOدر xv6 می باشد که در این حالت یک پردازه ایجاد شده و آماده اجرا نیست و برخی مقداردهی های اولیه هنوز انجام نشده اند و همان طور که اشاره شد پردازه ها در xv6 بلافاصله پس از allocproc وارد این وضعیت می شوند

●حالت ready :این وضعیت در عکس بالا زمانی رخ می دهد که پردازه آماده اجرا باشد و در انتظار گرفتن cpuباشد تا توسط scheduler به آن داده شود. این وضعیت مشابه وضعیت RUNNABLE در xv6 می باشد که پردازه ای آماده اجرا است و در صف اجرا شدن قرار دارد تا scheduler به آن cpu بدهد

●حالت running :این حالت در عکس بالا زمانی رخ می دهد که پردازه از زمان بند cpu را گرفته و در حال اجرا است. وضعیتی با همین نام یعنی RUNNING در xv6 وجود دارد

●حالت waiting :در این حالت پردازه صبر می کند تا فرایندی اجرا شود و خاتمه یابد مثلا نیاز به یک دستگاه i/o دارد بنابراین صبر می کند تا آن دستگاه کارش تمام شود. این حالت مشابه وضعیت SLEEPING در xv6 است که در آن پردازه منتظر یک واقعه یا منبع می ماند

●حالت terminated: در این حالت کار پردازه تمام شده و پردازه خاتمه می یابد و دیگر پردازه فعال نیست. این حالت مشابه وضعیت ZOMBIE در xv6 می باشد که پردازه ای exit را صدا کرده چون کارش تمام شده در حالی که پردازه والد هنوز منابع را آزاد نکرده یعنی wait را صدا نکرده است

سوال 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 xv6در این گذار از چه حالتی/حالت هایی به چه حالت/حالت هایی تغییر می کند؟ پاسخ خود را با پاسخ سوال۲ مقایسه کنید.

همان طور که اشاره کردیم با ساخته شدن پردازه، بلافاصله پس از خارج شدن از allocproc وضعیت پردازه EMBRYO است که معادل new است. حال باید به دنبال جایی بگردیم که این وضعیت را به RUNNABLE تغییر دهد. بعد از اینکه پردازه اول وارد init.c شد در ابتدا fork را صدا می زند. در تابع fork که در proc.c وجود دارد، بعد از گرفتن lock مربوط به ptable وضعیت پردازه به RUNNABLE تغییر می کند. همان طور که در سوال 2 اشاره کردیم وضعیت EMBRYO مشابه وضعیت new و وضعیت RUNNABLE همان وضعیت ready است بنابراین در این گذار، پردازه از EXMBYO به RUNNABLE می رود.

سوال 4-سقف تعداد پردازه‌های 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؟

ماکسیمم تعداد پردازه‌های ممکن در xv6 به شکل یک constant در فایل param.h تعریف شده است:

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

همانطور که می‌بینیم در تابع fork دستور allocproc صدا زده شده است:

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

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

تابع allocproc می‌بایست یک process جدید را اختصاص دهد. از آنجایی که process جدیدی در ptable.procموجود نیست

این تابع صفر را برمی‌گرداند. پس دستور fork نیز -1 را برمی‌گرداند که نشانه عدم موفقیت آن است.

این نکته باعث crash کردن سیستم نمی‌شود، بلکه kernel فعالیت خودش را برای بقیهprrocess های موجود ادامه می‌دهد.

```
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;
```

نهایتا user program باید مشابه زیر مقدار بازگشتی fork را جک کند. اگر منفی بود یعنی عملیات به درستی انجام نشده است:

```
pid = fork();
if (pid < 0) {
    printf("Fork failed: No available process slots\n");
} else if (pid == 0) {
    // This is the child process
} else {
    // This is the parent process
}
```

سوال 5- چرا نیاز است در ابتدای هر حلقه تابع scheduler، جدول پردازه‌ها قفل می‌شود؟ آیا در سیستم‌های تک‌پردازه‌ای هم نیاز است این کار صورت بگیرد؟

Ptable در واقع اطلاعات مربوط به پردازه‌های مختلف را در خود نگه می‌دارد. برای synchronization بهتر و حفظ data inconsistency و جلوگیری از بوجود آمدن race conditions بین پردازه‌های مختلف باید جدول پردازه‌ها را قفل کنیم.

در این حلقه ما به دنبال processهایی می‌گردیم که در RUNNABLE state باشند، اما ممکن است در بخش‌های دیگری از kernel، state این processها توسط دلایل زیر تغییر کند:

1. سیستم‌کال‌هایی مثل exit، kill، unused، wait می‌توانند state این process را به UNUSED, SLEEPING, RUNNABLE تغییر دهند.
2. Interruptها می‌توانند وضعیت سیستم را به RUNNING تغییر دهند.
3. در سیستم‌های چند هسته‌ای، می‌توانند scheduler مربوط به خودشان را اجرا کنند ولی با یک process table کار کنند.

به همین دلیل ptable را قفل می‌کنیم تا در فعالیت scheduler اختلال بوجود نیاید.

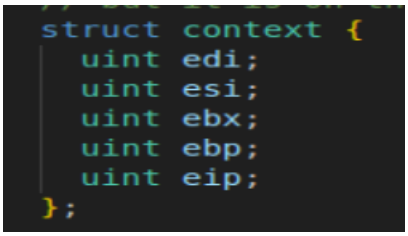
```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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;
```

از آنجایی که در سیستم‌های تک هسته‌ای فقط یک CPU core وجود دارد، فقط یک thread در هر لحظه در سیستم در حال اجراست و قرار نیست ptable توسط schedulerهای هسته‌های مختلف تغییر کند، ولی از آنجایی که ممکن است به خاطر وجود system callها و interruptها همچنان data inconsistency رخ دهد، بهتر است ptable قفل شود.

سوال 6- با فرض اینکه xv6 در حالت تک‌هسته‌ای در حال اجراست، اگر یک پردازنده به حالت runnable برود و صف پردازه‌ها در حال طی شدن باشد، در چه iterationای امکان schedule پیدا می‌کند؟

فرض کنیم که scheduler در حال iterateکردن روی ptable باشد و یک process که اول در وضعیت SLEEPING قرار داشته، توسط interrupt handler به وضعیت RUNNABLE می‌رود. Scheduler متوجه تغییر در وضعیت این process نمی‌شود و Iteration خود را ادامه می‌دهد و در iteration بعدی امکان schedule کردن آن پیدا می‌شود.

سوال 7- رجیستر های موجود در ساختار context را نام ببرید:



تصور بالا رجیسترهای استفاده شده در ساختار context را نشان می دهد (کد خام xv6 در این بخش مدنظر گرفته شده است).

حال به توضیح هریک از این موارد می پردازیم:

edi (Extended Destination Index)

این رجیستر معمولاً برای نگهداری آدرس مقصد در عملیات‌های کپی و مقایسه استفاده می‌شود.

esi (Extended Source Index)

این رجیستر معمولاً برای نگهداری آدرس منبع در عملیات‌های مشابه edi استفاده می‌شود.

ebx (Extended Base Register)

این رجیستر معمولاً به عنوان یک رجیستر پایه در آدرس‌دهی استفاده می‌شود.

ebp (Extended Base Pointer)

این رجیستر به طور معمول برای اشاره به بستر استک (stack frame) استفاده می‌شود.

eip (Extended Instruction Pointer)

این رجیستر نشان‌دهنده آدرس دستور بعدی است که باید اجرا شود.

سوال 8- مهمترین رجیستر قبل تعویض متن PC می باشد این رجیستر در ساختار context چه نام دارد و چگونه ذخیره می شود؟

رجیستر eip نمایانگر آدرس دستور بعدی است که باید اجرا شود، به این معنا که جریان کنترل برنامه را هدایت می‌کند.

نحوه ی تعویض متن:

وارد شدن به تابع زمان‌بندی:

هنگامی که یک وقفه زمانی (timer interrupt) رخ می‌دهد، کنترل به تابعی به نام scheduler منتقل می‌شود.

ذخیره وضعیت پروسه فعلی:

مقادیر رجیسترها از پروسه فعلی (پروسه‌ای که در حال اجراست) ذخیره می‌شود. این مقادیر در ساختار Process Control Block (PCB)پروسه فعلی ذخیره می‌شوند.

انتخاب پروسه جدید:

در تابعscheduler ، سیستم عامل پروسه جدیدی را برای اجرا انتخاب می‌کند. این انتخاب معمولاً بر اساس الگوریتم زمان‌بندی انجام می‌شود.

بازیابی وضعیت پروسه جدید:

پس از انتخاب پروسه جدید، مقادیر رجیسترها از PCB پروسه جدید بارگذاری می‌شوند. این بارگذاری شامل مقادیر ذخیره‌شده در PCB مربوط به پروسه جدید است.

تنظیم دستور اجرایی:

مقدار eip (مقدار نشان‌دهنده آدرس دستور بعدی) به آدرس شروع تابع پروسه جدید تنظیم می‌شود.

انتقال کنترل به پروسه جدید:

با بازیابی مقادیر رجیسترها و تنظیمeip ، کنترل به پروسه جدید منتقل می‌شود. این کار به وسیله یک پرش (jump)به آدرس مشخص‌شده توسط eip انجام می‌شود.

ادامه اجرای پروسه جدید:

پروسه جدید از همان نقطه‌ای که متوقف شده بود، ادامه می‌دهد و CPU به دستورات آن پرداخته و به طور معمولی ادامه می‌دهد (تا زمان تعویض متن بعدی که دوباره همین مراحل تکرار می شود).

سوال 9- در ابتدای تابع scheduler ایجاد وقفه به کمک تابع sti انجام می‌شود. اگر وقفه‌ها فعال نمی‌شد چه مشکلی بوجود می‌آمد؟

دلیل فعال‌سازی دوره‌ای وقفه‌ها روی یک پردازنده این است که ممکن است هیچ فرایند RUNNABLEای وجود نداشته باشد، زیرا فرایندها (مثلاً shell) منتظر ورودی/خروجی (I/O) هستند؛ اگر scheduler وقفه‌ها را همیشه غیرفعال نگه دارد، ورودی/خروجی هرگز نمی‌رسد.

سوال 10-بنظر شما وقفهٔ تایمر هر چه مدت یک بار صادر می شود؟

هر 10 میلی ثانیه

سوال 11- چه تابعی منجر به انجام شدن گذار interrupt می‌شود؟

زمانی که یک interrupt اتفاق می‌افتد، تابع ()trap از interrupt handler صدا زده می‌شود.

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

داخل تابع trap، اگر process همچنان در حال اجرا بود، تابع yield صدا زده می‌شود که state آن process را به ready تغییر می‌دهد و با استفاده از scheduler، process بعدی را انتخاب می‌کند.

```
// Give up the CPU for one scheduling round.
void yield(void)
{
  acquire(&ptable.lock); // DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

پس در واقع تابع yield سبب این گذار می‌شود.

سوال 12- کوانتوم زمانی این پیاده سازی از زمان بند نوبت گردشی چند میلی ثانیه است؟

در این سیستم‌عامل، کوانتوم زمانی برای الگوریتم نوبت چرخشی با توجه به وقفه تایمر تعیین می‌شود. در سوال ۱۰ به دست آمد که وقفه تایمر ۱۰ میلی ثانیه است، یعنی هر ۱۰ میلی ثانیه یکبار صدا زده می‌شود، پس کوانتوم زمانی نیز ۱۰ میلی ثانیه است.

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

در فایل proc.c تابع wait وجود دارد. این تابع به این صورت کار می کند که ابتدا به در ptable به دنبال پردازه های فرزندی می گردد که exit را صدا زده اند. سپس چک می کند آیا فرزندی وجود دارد که exit را صدا کرده یا خیر. اگر پردازه این چنینی وجود نداشته باشد، lockمربوط به ptable را آزاد می کند. در غیر این صورت به کمک تابع sleep صبر می کند تا فرزند exit کند.

سوال 14-با توجه به پاسخ سوال قبل، استفاده های دیگر این تابع چیست؟

به طور کلی می توان گفت از تابع sleep برای بلاک کردن پردازه ها می توان استفاده کرد چرا که پردازه به وضعیت SLEEPING می رود و دیگر توسط زمان بند به آن cpu تعلق نمی گیرد. بنابراین استفاده اصلی sleepبلاک کردن پردازه تا زمانی است که یک شرطی برقرار شود یا اتفاق رخ دهد.

برای مثال در تابع consoleread از Sleep استفاده شده تا پردازه فعلی را تا زمانی که داده در بافر ورودی بیاید بلاک کند. در این تابع ابتدا خالی بودن بافر بررسی می شود که اگر بافر خالی بود sleep را صدا می زند

سوال 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 به نام wakeup وجود دارد که اگر شرط برقرار شد یا اتفاق که منتظر بودیم رخ داد، پردازه را بیدار کند.

سوال 16- با توجه به سوال 9، تابع sti سبب چه گذاری می‌شود؟

گذار از waiting state به ready state زمانی اتفاق می‌افتد که eventای رخ دهد یا یک عملیات IO کامل شود. طبق توضیحات ارائه شده در سوال 9، تابع sti سبب این گذار می‌شود.

سوال 17- آیا تابع دیگری وجود دارد که سبب این گذار شود؟

تابع wakeup برای بیدارکردن processهایی استفاده می‌شود که در waiting state در حالت SLEEPING هستند. زمانی که یک process منتظر یک event خاص یا اتمام یک عملیات IO است، در state waiting می‌ماند. با رخداد آن event، این تابع صدا زده می‌شود و processهای مروبطه به ready state می‌روند.

```
// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

تابع exit اجازه می‌دهد که یک process از سیستم خارج شود و در موقعیت‌هایی سبب wakeup روی processهایی می‌شود که در انتظار آن process بوده‌اند.

علاوه بر این‌ها، توابعی هستند که بعد از تمام شدن یک عملیات IO، process ای که منتظر آن عملیات IO بوده را wakeup می‌کنند و آن را به ready state منتقل می‌کنند، مثل تابع ioqueue_dequeu.

سوال 18- رویکرد xv6 دربرابر فرآیندهای orphan چیست؟

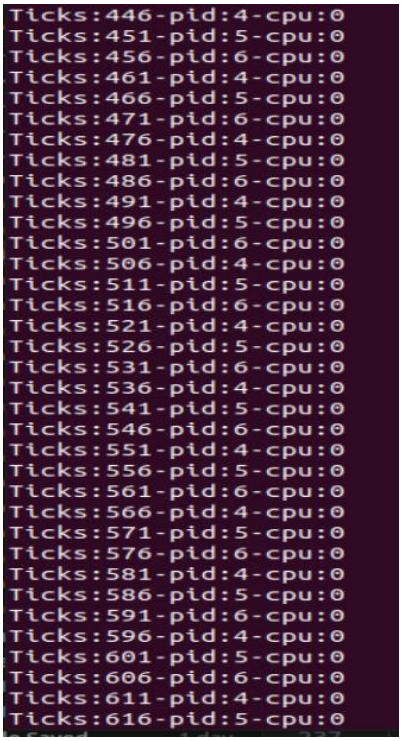
تعریف: زمانی که یک فرایند والد پیش از فرزند خود خاتمه یابد، فرزند به یک فرآیند یتیم (Orphan) تبدیل می‌شود.

درxv6 ، وقتی یک فرایند یتیم می‌شود، فرایند init (فرایند شماره 1) به عنوان والد جدید آن فرایند عمل می‌کند. این کار به مدیریت بهتر منابع و جلوگیری از نشت حافظه کمک می‌کند چرا که از این طریق منابع به درستی آزاد می شوند و همچنین wait فرایند init می‌تواند به انتظار فرآیندهای یتیم بنشیند تا به اتمام برسند و وضعیت اتمام آن‌ها را دریافت کند.

سوال 19-مقدار CPUS را مجددا به 2 برگردانید. آیا همچنان همان ترتیب مشاهده می شود ؟ علت چیست؟

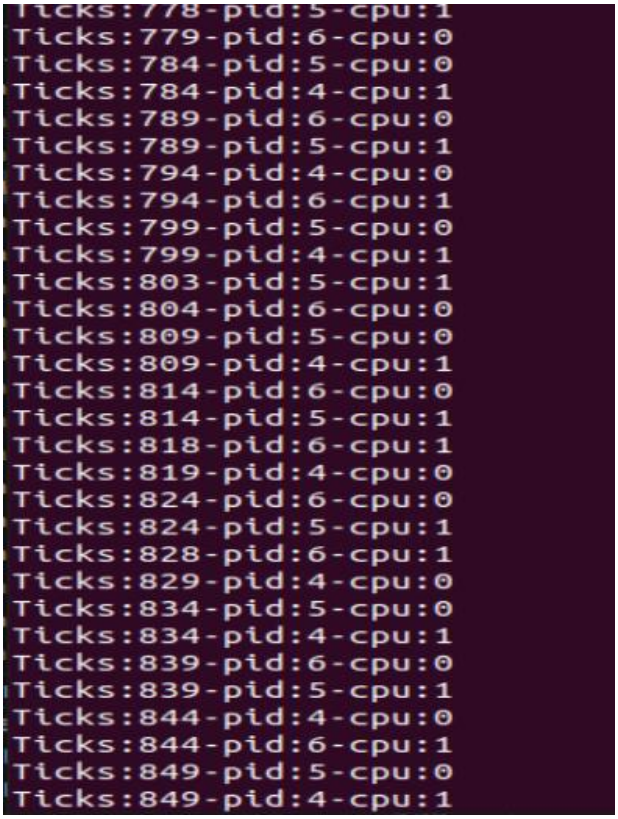
خیر این ترتیب دوباره تکرار نمی شود دلیل این موضوع این است که در این حالت به نحوی می توان گفت که ما در حالت multicore هستیم پس برای هر پردازنده به صورت مستقل الگوریتم Round Robin اجرا می شود این یعنی در هرزمان که یکی از cpu ها توان انجام عملیاتی را داشت از روی ready queue یک پردازه قابل اجرا برداشته و آن را اجرا می کند این یعنی اگر ما 3 پردازه 4، 5 و 6 را داشته باشیم (که اعداد نشان دهنده pid آن ها باشد) همانطور که مشخص است در حالت داشتن چند پردازنده لزوما این ترتیب Round Robin ما برای اجرا نخواهد بود و هر ترتیبی ممکن است.

حالت زیر داشتن تک پردازه ای را نشان می دهد:



همانطور که به چشم می خورد الگوریتم درحال اجرای Round Robin برروی یک cpu است که ترتیب ورود به صف ذکر شده و این ترتیب به هم نمی خورد مگر آنکه یکی از پردازه ها تنها در سیستم باقی مانده و تمام اسلایس های زمانی بعدی را به خود اختصاص دهد.

در حالت چند پردازه ای این ترتیب به هم می خورد یعنی دیگر نمی توان انتظار داشت که ترتیب 4,5,6 حفظ شود و در عکس زیر بخشی از این خروجی آمده است:



هر cpu یک Round Robin مخصوص خود برروی Ready queue اجرا می کند پس بدیهی است که ترتیب چاپ شده تغییر کند.

سوال 20- در صورت نیاز به مقدار دهی اولیه به فیلدهای اضافه شده در ساختار cpu در چه تابعی بهتر است این کار صورت بگیرد؟

ما در این پروژه این کار را در تابع ()mpmain در فایل main.c انجام دادیم دلیل انتخاب این تابع عبارت است از :

تابع ()mpmain وظیفه راه‌اندازی هر پردازنده به صورت مستقل را بر عهده دارد. از آنجایی که هر پردازنده در این تابع مقداردهی و آماده می‌شود، افزودن مقداردهی فیلدهای اختصاصی CPU به این نقطه طبیعی و منطقی است، زیرا در اینجا دسترسی مستقیم به ساختار CPU مربوط به پردازنده فعلی داریم.

این تابع زمانی اجرا می‌شود که سایر مراحل اساسی سیستم مثل شناسایی پردازنده‌ها و تنظیمات پایه‌ای cpu قبلاً انجام شده‌اند، در این مرحله، پردازنده‌ها در حال نهایی‌سازی راه‌اندازی خود هستند، بنابراین هرگونه مقداردهی در اینجا بدون تداخل با سایر مراحل انجام خواهد شد.

با توجه به اینکه هر پردازنده به صورت شخصی مقداردهی می شود از بروز race condition نیز جلوگیری می کند و هر مقدار دهی به صورت مستقل رخ می دهد.

این تابع معمولاً پیش از شروع زمان‌بند اجرا می‌شود. این باعث می‌شود تمامی فیلدهای جدید قبل از استفاده از پردازنده‌ها در اجرای فرآیندها، مقداردهی شوند و از هرگونه رفتار پیش‌بینی‌نشده در زمان‌بند به دلیل مقداردهی‌نشده بودن فیلدها جلوگیری شود.

سوال 21- با توجه به سیاست های پیاده سازی شده در سطح های دوم و سوم و همچنین استفاده از روش time-slicing توجیه کنید چرا همچنان مشکل starvation امکان رخ دادن دارد؟

با توجه به اینکه ما به دو علت از یک صف خارج می شویم

- به دلیل تمام شدن پردازه های داخل صف
- تمام شدن تایم زمانی

این شرایط را درنظر بگیریم که در صف اول ما پردازه های پر اولویت قرار دارند و ما به خاطر یکی از دلایل ذکر شده در بالا cpu را تسلیم کرده و به صف بعدی می رویم و در این صف هیچ پردازه ای وجود نداشته و درنهایت به صف FCFS می رسیم حالا در این صف تمامی user program ها قرار دارند (با توجه به فرض ذکر شده در صورت پروژه) اما کوانتوم زمانی متعلق به این صف تنها 100 میلی ثانیه است که این یعنی ممکن است بخش اعظمی از اعضای این صف cpu را هرگز در اختیار نگیرند (با فرض اینکه یک برنامه cpu bound بوده و به دنبال I/O نرود) حال پس از تمام شدن این کوانتوم زمانی باز به صف اول بازگشته و پردازه های پر اولویت اجرا می شوند .

همانطور که اشاره شد کوانتوم زمانی مربوط به این صف 300 میلی ثانیه و کوانتوم زمانی صف دوم برابر 200 میلی ثانیه می باشد پس در حقیقت قبل اینکه ما به صف سوم برسیم در بدترین حالت 500 میلی ثانیه صبر کرده ایم و این امکان وجود دارد که یک پردازه که در انتهای صف سوم قرار می گیرد هرگز cpu را دریافت نکند پس ما نتوانستیم با این روش starvation را حل کنیم.

بلکه نیاز به مکانیزمی داریم که اولویت پردازه ها را افزایش داده تا سهم زمانی بیشتری از این cpu دریافت کنند.

سوال 22- به چه علت زمانی که پردازه در حالت sleeping به سر می برد در وضعیت waiting محاسبه نمی شود؟

دلیل این موضوع این است که مدت زمان انتظار مربوط به زمانی می شود که پردازه درون صف ready به سر می برد و منتظر است تا cpu به آن اختصاص یابد.

استیت sleep به طور کامل با این استیت تفاوت دارد دلیل این موضوع این است که پردازه ای که در استیت sleep قرار گرفته است منتظر یک اتفاق یا یک منبع خاصی است به طور مثال timer, signal, I/O completion اتفاق بیفتد تا به ادامه کار بپردازد در چنین حالتی پردازه مد نظر واجد شرایط برای دریافت cpu نیست (یعنی در حقیقت در آماده برای اجرا نیست پس schedule کردن آن معنا ندارد).

به همین دلیل پردازه ای که در استیت ready قرار دارد به استیت wait منتقل می شود و منتظر می ماند تا یک اتفاق خاص یا سیگنال خاصی دریافت شود و سپس به صف ready باز گردد.

از آن جایی که در استیت sleep برای گرفتن cpu در رقابت شرکت نمی کند و توسط scheduler درصدد انتخاب شدن قرار نمی گیرد پس این تایم جزو زمان انتظار حساب نمی شود.

به همین دلیل زمان انتظار را زمانی بیان می کنیم که پردازه درون ready queue به سر میبرد و واجد شرایط برای دریافت cpu می باشد و این زمان انتظار ناشی از الگوریتم زمانبندی cpu ایجاد می شود.