OS_LAB_3

محمد مهدی خصالی 810100134

علی داداشی 810100138

محمد مهدی داورزنی 810100140

سوال 1:

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

همچنین پردازه ها در این پردازه میتوانند این وضعیت ها را داشته باشند

- Running : پردازه ای در حال حاضر در حال اجراست
- Sleeping : پردازه ای که منتظر یک رخداد (مانند ورودی و خروجی) برای ادامه کار خود است
- Runnable : پردازه ای که میتواند شروع به اجرا کند اما در حال حاضر CPU درگیر است
- Stopped : پردازه ای که به طور موقت متوقف شده (به دلیل دریافت یک سیگنال خاص و یا ...)

- Zombie : پردازه ای که متوقف شده اما هنوز در جدول داده ها دارای داده است زیرا پردازه والد از دستور ()wait استفاده نکرده است .

شباهت با شکل 3.3 کتاب منبع :

- Process State -> متغیر state در proc
- Process number -> متغیر pid در proc
- List of open files -> متغیرofile در proc
- Memory limits -> متغیر sz در proc

Registers -> متغیر context درproc

سوال 2:

شباهت با شکل یک :

- Running معادل running در شکل یک
- Sleeping معادل waiting در شکل یک
- Runnable معادل ready در شکل یک
- Stopped معادل به نوعی waiting در شکل یک
- Zambie به نوعی معادل terminate در شکل یک

سوال3:

تابع ()allocproc اینکار را انجام می دهد . این تابع در جدول پردازه ها یک پردازه ایجاد می کند و مقادیر ابتدایی را به آن می دهد و آماده ی اجرایش می کند (ناقص)

سوال 4:

حداکثر تعداد پردازه ها به اندازه سایز Process Table است . سایز این جدول با استفاده از متغیر NPROC مشخص می شود که به طور پیش فرض برابر 64 است . اگر تعداد پردازه ها از این عدد تجاوز کند دیگر پردازه ای ایجاد نمی شود تا زمانیکه پردازه ها به اتمام برسند (در واقع کرنل ارور می دهد)

همچنین برنامه سطح کاربر از failure , system call دریافت می کند (معمولا عدد -1)

سوال 5:

سه دلیل قفل کردن:

به طور کلی چون در سیستم های multiprocessor چندین پردازنده می توانند به جدول پردازه ها دسترسی داشته باشند , فرآیند قفل کردن از رخداد هایی مانند تغییر غیر عمدی داده ها و race condition و ناهماهنگی بین پردازنده ها جلوگیری می کند .

در سیستم های تک پردازنده ای می تواند مفید باشد اما ضروری نیست زیرا ریسک تغییر داده و یا پردازه های وجود ندارد

سوال 6:

از آنجا که تابع schedular جدول پردازه ها را طی می کند تا به یک پردازه Runnable برسد

اگه تغییر وضعیت قبل از رسیدن تابع به این پردازه باشد در همان iteration اجرا می شود وگرنه در iteration بعدی .

سوال 7:

رجیسترها به شرح زیر است :

- Edi
- Esi
- Ebx
- Ebp
- Eip

سوال 8:

Program Counterدر ساختار XV6 همان eip است

این رجیستر قبل از انجام عملیات تعویض آدرس next instruction را در خود ذخیره میکند تا هر زمان خواست ادامه دهد از همین نقطه شروع کند

سوال 9:

اگر وقفه ها فعال نشود امکان ایجاد چندین مشکل وجود دارد که به شرح زیر است :

- Preemption دیگر قابل پیاده سازی نبود تا پردازه ها جایگزین یکدیگر شوند

- System calls and hardware intrrupts عملا پیاده سازی نمی شدند
- در مواقعی که نیاز به interrupt برای ادامه هست سیستم دچار hang می شود چون برای ادامه به intrupt نیاز دارد اما در دسترس نیست

سوال 10:

به صورت فرکانس پیش فرض timer intrupts برابر 100Hz می باشد . یعنی هر ثانیه 100 بار تکرار می شود . در واقع در فایل trap.c قابل configure کردن است

سوال 11:

تابع trap این کار را میکند:

System Call Handling:

```
if(tf->trapno == T_SYSCALL){
  if(myproc()->killed)
    exit();
  myproc()->tf = tf;
  syscall();
  if(myproc()->killed)
    exit();
  return;
}
```

Re-check Process State:

```
if(myproc() && myproc()->killed && (tf->cs & 3) == DPL_USER)
  exit();
```

Timer Interrupt:

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

سوال12:

حدودا 10ms.

سوال 13:

برای منتظر ماندن از تابع ()sleep استفاده میکند. تابع ()sleep شرطی است یعنی تا وقتی که فرایند فرزند در حال اجرا است, اجرای برنامه اصلی را متوقف می‌کند.

در حقیقت یک نوع interrupt است. پس از اینکه فرزند سیگنالی به عنوان پایان کار خود ارسال کند، ()sleep اجرا میشود.

سوال14:

به عنوان نمونه در pipe از ()sleep استفاده میشود. وقتی بافر یک پایپ پر میشود تابع ()sleep توسط فرستنده فراخوانی میشود. این امر سبب میشود تا یک گیرنده (خواننده) از باقر مقداری بردارد و فرستنده متوجه شود که بافر جای خالی دارد. در حقیقت نوشتن در بافر وابسته به یک شرط است و آن هم مصرف شدن از buffer است.

سوال15:

در سطح kernel تابع wakeup را داریم.

تابع wakeup: این تابع به event های مربوطه گوش می‌دهد و سپس فرایند را از حالت wating در حالت قابل اجرا میگذارد. به این صورت یک process از event مطلع میشود.

سوال16:

از حالت انتظار (waiting) به حالت ready تغییر میکند.

سوال 17:

بله وجود دارد. با استفاده از تابع yield این امر ممکن است. این تابع می‌تواند process های در حال اجرا را برای بعدا برنامه ریزی کنند یعنی از استیت running به runnable تغییر وضعیت دهد.

در حقیقت yield به صورت مکرر و دوره ای اجرا میشود و مانع این میشود که پردازنده یک کار طولانی را انجام ندهد (اشغال نکند) و در اجرای فرایند های طولانی (از نظر زمانی) cpu را کنترل می‌کند و به طور مساوی به فرایند ها اختصاص می‌دهد.

سوال 18:

Init process: یک فرایندی ست که به صورت دوره ای اجرا میشود و اصطلاحا wait می‌کند

وقتی process پدر تمام میشود (و CPU از آن گرفته می‌شود) init process وارد کار می‌شود و اون process هایی که پدر نداشتند را تحت کنترل خود قرار میدهد و در نهایت از source leaking جلوگیری می‌کند .

این روش باعث میشود تا فرایند های بدون پدر تا بینهایت باقی نماند و بالاخره خاتمه یابد.

Round Robin

در ابتدا یک برنامه کاربر ایجاد میکنیم که یک فرزند ایجاد میکند و به صورت همروند با فرزند اجرا میشود. منتها پراسس فرزند Child چاپ میکند و پروسس والد Parent. پر واضح است که تعداد چاپ Child پشت سر هم و تعداد چاپ Parent پشت سر هم بیشتر میشود اگر کوانتوم زمانی بیشتر شود. نتیجه تست این برنامه و کد آن به صورت زیر خواهد بود:

```
Child : 930
ChilParent : 879
Parent : 880
Parent d : 931
Child : 932
Child : 93: 881
Parent : 882
Parent : 883
Pare3
Child : 934
Child : 935
nt : 884
Parent : 885
Parent : 8Child : 936
Child : 937
```

RoundRobinTest.c > main(int, char * [])

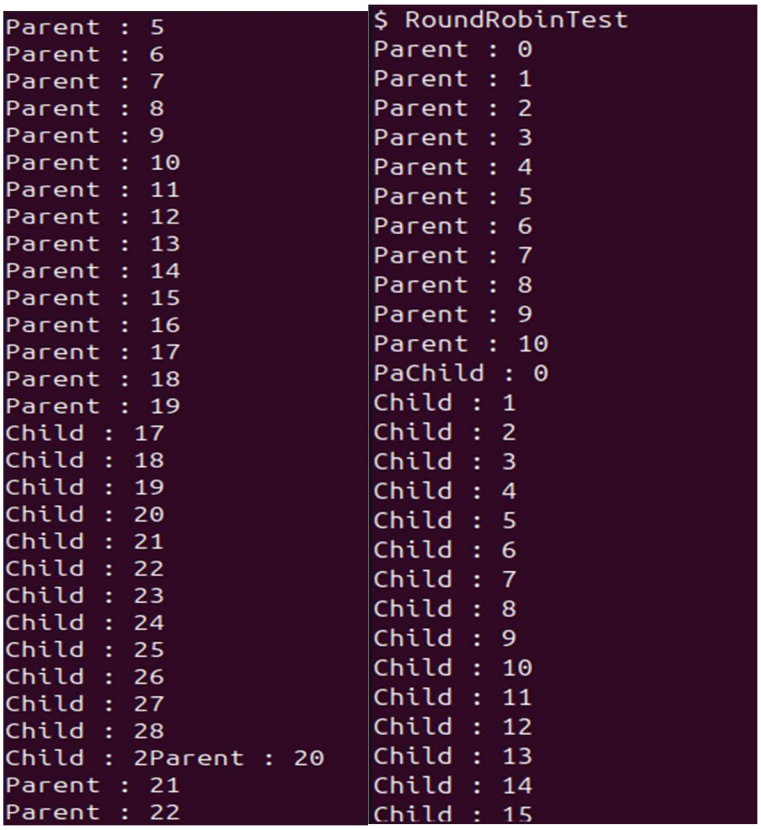
```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fs.h"

int main(int argc, char* argv[])
{
    int Pid = fork();

    if(Pid < 0)
    {
        printf(1, "Failed to create child process!\n");
    }
    else if(Pid == 0)
    {
        for(int i = 0; i < 1000; i++)
        {
            printf(1, "Child : %d \n", i);
        }
        exit();
    }
    else
    {
        for(int i = 0; i < 1000; i++)
        {
            printf(1, "Parent : %d \n", i);
        }
        exit();
    }

    return 0;
}
```

حال در تابع trap در فایل trap.c که تعویض متن در آن صورت میگیرد، بین خط 106 و 107 این شرط را اضافه میکنیم که کوانتوم زمانی مضربی از یک عدد باشد برای مثال 10 یا 5:



در نهایت کوانتوم زمانی را روی 5 تنظیم کردیم.

سوال 19:

وقتی که مقدار CPUS را عدد 2 قرار دهیم، پروسس پدر و فرزند هر کدام روی یک هسته مجرا و به صورت هم روند اجرا میشوند. و این موضوع باعث در هم ریختگی شدیدتری در مقادیر چاپ شده میشود. چرا که دیگر در هر لحظه هر دو پروسس در حال اجرا هستند و به طور موازی روی کنسول چاپ میکنند. مشاهده میشود که با تغییر مقدار کوانتوم زمانی این وضعیت تغییری نمیکند.

SJF

باید زمان پیش بینی شده و احتمال آنرا به استراکچر proc در proc.h اضافه کنیم و در تابع allocproc مقداردهی اولیه کنیم.

برای تست این الگوریتم زمانبندی نیز باید وقفه های زمانی را غیر فعال کنیم. زیرا این الگوریتم هیچ وقفه زمانی ای ندارد و فقط کوتاه ترین کارها را ابتدا انجام میدهد. سپس در تابع برنامه ریزی باید پروسس ها را بر اساس زمان خاتمه مرتب کنیم. برای این کار میتوان یک تابع سورت نوشت.

```
C proc.c > sys_print_info(void)
80   allocproc(void)
95     p->state = EMBRYO;
96     p->pid = nextpid++;
97     p->BurstTime = 2;
98     p->Confidence = 50;
```

```
{
  struct proc *p;
  struct proc *arg_min;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state != RUNNABLE || p->QueueNumber != SJF)
      continue;
    arg_min = p;
    for (struct proc *i = p + 1; i < &ptable.proc[NPROC]; i++)
    {
      if (i->state != RUNNABLE || i->QueueNumber != SJF)
        continue;
      if (arg_min->BurstTime > i->BurstTime && arg_min->pid != i->pid)
      {
        arg_min = i;
      }
    }
    if (arg_min != p)
    {
      // cprintf("Swapped %d with %d\n", p->pid, arg_min->pid);
      // cprintf("state pid %d: %d\n", p->pid, p->state);
      // cprintf("state pid %d: %d\n", arg_min->pid, arg_min->state);
      struct proc temp = *p;
      *p = *arg_min;
      *arg_min = temp;
    }
  }
}
```

در این شرایط اگر متغیری تحت عنوان احتمال مطرح نبود، کافی بود پروسس ها را به ترتیب مرتب شده پیمایش و اجرا کنیم. اما حالا که با این پدیده روبرو هستیم، عددی تصادفی ایجاد کرده و با مقدار احتمال مقایسه میکنیم. اگر کمتر بود که پروسس را اجرا میکنیم و اگر بیشتر بود از آن گذر کرده و این عملیات را روی پروسس بعدی اجرا میکنیم.

```
void scheduler(void)
  for (;;)
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      int random = ticks * (p->pid + 1) + ((int)p->name[0] + 1) * 357 + 666;
      random = ((random * random) % 100);
      // cprintf("Random number: %d for pid: %d\n", random, p->pid);
      if (random > p->confidence)
        continue;
```

:FIFO

درست مثل قسمت sjf، در این قسمت نیز نیازی به فراخوانی تابع yield() نداریم. بایستی در ابتدای ایجاد پروسس زمان ورود را مقدار دهی کرده و با هر بار فراخوانی تابع برنامه ریز یا schedule پروسسی که زمان ورود آن از بقیه کمتر است را اجرا میکنیم و در پایان پروسس تابع exit() را فراخوانی میکنیم.

Aging:

پیاده سازی این قسمت میتواند به این صورت باشد که در برای استراکت proc یک عضو age در نظر میگیریم و هر پروسسی که در وضعیت یا استیت runnable بود در وقفه زمانبند، به اندازه یکی به age اضافه شود. همچنین در زمان تغییر صف یک پروسس لازم است age به مقدار صفر تنظیم شود.

```
732  void update_age(void)
733  {
734    struct proc *p;
735    acquire(&ptable.lock);
736    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
737    {
738      if (p->state != RUNNABLE || p->pid < 3)
739      {
740        p->Age = 0;
741        continue;
742      }
743      p->Age++;
744    }
745    release(&ptable.lock);
746  }
747
```

```
void update_queue_number(void)
{
  acquire(&ptable.lock);
  for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    // TODO 80 -> 800
    if (p->QueueNumber == RR || p->Age < 80 || p->pid < 3)
      continue;
    p->QueueNumber++;
    p->Age = 0;
    p->EnterTime = ticks;
    cprintf("Process %d moved to queue %d\n", p->pid, p->QueueNumber);
  }
  release(&ptable.lock);
}
```

تغییر سطح:

در هر بار timer interrupt سن پروسس ها را چک میکنیم و پروسسی که سن آن از 80 بیشتر بود را به سطح بالاتر انتقال میدهیم.

```
// 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)
     {
        update_age();
        update_queue_number();
        mycpu()->ticks++;
        int time = mycpu()->ticks;
```

Time slicing:

ابتدا به استراکت cpu به ازای هر صف یک متغیر اضافه میکنیم که نشان دهنده زمان باقی مانده است و هر کوانتوم زمانی خود را 100 میلی ثانیه در نظر میگیریم و به ازای هر کوانتوم یک واحد از زمان هر کدام را کم میکنیم. زمانها طبق خواسته مساله 2، 3 و 1 است. هر زمان که دیدیم زمان هر کدام تمام شده بعدی را مقدار دهی میکنیم و برای راند روبین علاوه بر این کار پروسس در حال اجرا را هم ذخیره میکنیم، ایضا در تابع اسکجول هم نسبت به اینکه در کدام هستیم همان را برنامه ریزی میکنیم. نکته دیگر اینکه اگر در راند روبین بودیم، به ازای هر 5 کوانتوم زمانی مجدد راند روبین را صدا میکنیم.

```
void scheduler(void)
{
  for (;;)
  {
    sti();

    acquire(&ptable.lock);
    handle_change_queue();

    if (mycpu()->RR > 0)
      RR_scheduler();
    else if (mycpu()->SJF > 0)
      SJF_scheduler();
    else if (mycpu()->FCFS > 0)
      FCFS_scheduler();

    else
      panic("No queue selected");

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

```
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
```

اتمام پروسس زودتر از اتمام برش زمانی:

گاهی اوقات پیش می آید که کار یک صف زودتر از اتمام برش زمانی مربوط به آن تمام میشود، در این حالت نباید این مقدار را به صف بعدی داد. برای هندل کردن این مساله باید برای هر کلاک، تایمر جدا در نظر بگیریم که در حالت عادی مانند ساعت اصلی بالا میرود، اما زمانی که در وسط کار صف به پایان میرسد، زمان آن صفر میشود که باقی مانده زمان به صف دیگری تخصیص داده نشود یا از زمان صف دیگری چیزی کم نشود.

سوال 20:

در فایل main.c به ازای هر ویژگی که به پردازنده اضافه میشود میتوان در فایل vm.c برای آن یک تابع مقدار دهی اولیه نوشت و آن را تعریف و در تابع main درفایل main.c صدا زد.

سوال 21:

از آنجا که روش تایم اسلایس یا برش زمانی برای جابجایی در بین صف ها مشکل قحطی زدگی یا استارویشن را در هیچ کدام از صف ها که روش آنها دارای استارویشن است، حل نمیکند، ما از روش aging یا سالمند شدن استفاده میکنیم و بدین ترتیب پروسس ها را به سطوح بالاتر میبریم تا نهایتا به سطح اول یا راند روبین برسند که مشکل قحطی زدگی ندارد.

سوال 22:

پروسس مورد نظر در این حالت اصلا آمادگی اجرا شدن ندارد و منتظر رویدادی است که آنرا به صف آماده ببرد. پس این بازه زمانی را نمیتوان جزو زمان انتظار معطلی برای یک پروسس در نظر گرفت و بالا رفتن در صف به علت چنین انتظاری تفاوتی به حال پروسس ایجاد نمیکند چون هنوز شرایط اجرا شدن ندارد.