به نام خدا

علی شیخ عطار 99542222
صبا کیانوش 400522157

فعالیت: تابع scheduler و sched و بررسی ساختار ptable

ساختار ptable

ابتدا ساختار ptable را بررسی می‌کنیم. این ساختار یک جدول فرایندها است که تمامی فرایندهای سیستم را در بر می‌گیرد.که به صورت زیر است:

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

lock: قفل چرخان (spinlock) برای محافظت از دسترسی همزمان به ptable.

proc: آرایه‌ای از ساختارهای proc که هر کدام نماینده یک فرایند هستند. تعداد این فرایندها برابر با NPROC است.

```
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;){
    sti();
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);
  }
```

تابع scheduler:تابع scheduler در xv6 یک الگوریتم زمانبندی ساده و مؤثر است که از روش چرخشی (round-robin) استفاده می‌کند. این تابع با بررسی مداوم جدول فرایندها، فرایندهای قابل اجرا را پیدا کرده و آن‌ها را به طور منصفانه به CPU اختصاص می‌دهد. سادگی این طراحی آن را به ابزاری عالی برای یادگیری مفاهیم اساسی زمانبندی فرایندها و سوئیچ زمینه در سیستم‌عامل‌ها تبدیل کرده است.
حلقه بی‌نهایت باعث می‌شود که زمانبند به طور مداوم فرایندها را بررسی کند.دستور sti وقفه ها را روی پردازنده فعال می‌کند، بنابراین زمانبند می‌تواند در صورت لزوم متوقف شود (مثلاً توسط یک وقفه تایمر).سپس زمانبند قفل ptable را می‌گیرد تا دسترسی همزمان به جدول فرایندها را کنترل کند.سپس حلقه ای زده می شود که همیشه درحال پیدا کردن یک فرایند به حالت RUNNABLE است.

حال وقتی فرآیند مورد نظر پیدا شد متغیر global فرآیند را(c.proc) را برابر فرآیند پیدا شده میگذاریم. با دستور switchuvm(P) در واقع ادرس page table فرآیند مورد نظر را به آدرس قابل دسترسی cpu می دهیم بعد وضعیت فرایند را به RUNNING تغییر می دهیم و دوباره باید دسترسی به حافظه را به کرنل بازگردانیم.
تابع swtch مسئول انجام یک تعویض زمینه است که شامل ذخیره وضعیت فعلی CPU (زمینه اجرای فعلی) و بازگرداندن وضعیت یک زمینه اجرای دیگر می‌شود.پس از اتمام زمان فرایند یا متوقف شدن آن، متغیر proc به 0 تغییر میکند.در نهایت، زمانبند قفل ptable را آزاد می‌کند تا دیگر عملیات ها بتوانند به جدول فرایندها دسترسی داشته باشند.

در زمان بوت سیستم bootloader اجرا میشود و کرنل در مموری لود میشود.

کرنل بعد از لود شدن، زیرسیستم های مختلفی مثل process management, memory management و hardware device را initialize میکند.
در همین حین، cpu هارو initialize میکند و به تبع آن تابع scheduler اجرا میشود.
به وسیله ی تابع mpmain، تابع scheduler برای هر cpu اجرا میشود.

```
int main(void) {
    ...
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc();      // kernel page table
    mpinit();        // collect info about this machine
    lapicinit();     // interrupt controller
    seginit();       // segment descriptors
    picinit();       // disable pic
    ioapicinit();    // another interrupt controller
    consoleinit();   // console hardware
    uartinit();      // serial port
    pinit();         // process table
    tvinit();        // trap vectors
    binit();         // buffer cache
    fileinit();      // file table
    ideinit();       // disk
    if(!ismp)
        timerinit(); // uniprocessor timer
    userinit();      // first user process
    mpmain();
}
```

```
// Common CPU setup code.
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
}      You, last month • Initial XV6
```

در سه صورت تابع schedule در ادامه ی کار صدا زده میشود و scheduling صورت میگیرد:

۱- فرایند بلاک شود(منتظر I/O باشد.)
۲- فرایند به صورت اختیاری cpu را واگذار کند.
۳- فرایند کوانتوم مربوط به خود را مصرف کند و در این صورت نیز قبضه میشود و پردازنده به فرایند ی runnable بعدی اختصاص داده میشود.

Timer در XV6 تابع مجزایی ندارد و در چند فایل این مفهوم پیاده سازی شده است که مهم ترین آن ها lapic.c و trap.c می باشد.
مهم ترین وظیفه ی Timer، راه اندازی و سازماندهی سخت افزار برای timer و setup کردن interrupt handling می باشد.
رسیدگی به timer interrupt ها بر عهده ی تابع timerintr در فایل trap.c می باشد.
هندل کردن time-based event ها توسط همین تابع صورت میگیرد.
در زمان شروع سیستم و اجرای تابع main و غیره، timer شروع به کار میکند و در ابتدا local APIC و بقیه ی سخت افزار های مربوط به تایمر را راه اندازی میکند.
lapicinit وظیفه ی راه اندازی local APIC یا همان LAPIC timer را دارد که وظیفه ی این تابع، تولید interrupt های دوره ای در سیستم های چندپردازنده ای می باشد و همچنین نرخ تایم را مقدار دهی میکند (مقدار بایدیفالت آن ۱۰ میلیون می باشد).

```
void
lapicinit(void)
{
  if(!lapic)
801027f0: a1 80 16 11 80        mov    0x80111680,%eax
801027f5: 85 c0                 test   %eax,%eax
801027f7: 0f 84 cb 00 00 00     je     801028c8 <lapicinit+0xd8>
  lapic[index] = value;
801027fd: c7 80 f0 00 00 00 3f  movl   $0x13f,0xf0(%eax)
80102804: 01 00 00
  lapic[ID];  // wait for write to finish, by reading
80102807: 8b 50 20              mov    0x20(%eax),%edx
  lapic[index] = value;
8010280a: c7 80 e0 03 00 00 0b  movl   $0xb,0x3e0(%eax)
80102811: 00 00 00
  lapic[ID];  // wait for write to finish, by reading
80102814: 8b 50 20              mov    0x20(%eax),%edx
  lapic[index] = value;
80102817: c7 80 20 03 00 00 20  movl   $0x20020,0x320(%eax)
8010281e: 00 02 00
  lapic[ID];  // wait for write to finish, by reading       You, las
80102821: 8b 50 20              mov    0x20(%eax),%edx
  lapic[index] = value;
80102824: c7 80 80 03 00 00 80  movl   $0x989680,0x380(%eax)
8010282b: 96 98 00
  lapic[ID];  // wait for write to finish, by reading
8010282e: 8b 50 20              mov    0x20(%eax),%edx
  lapic[index] = value;
80102831: c7 80 50 03 00 00 00  movl   $0x10000,0x350(%eax)
80102838: 00 01 00
  lapic[ID];  // wait for write to finish, by reading
8010283b: 8b 50 20              mov    0x20(%eax),%edx
```

```
void
lapicinit(void)
{
  if(!lapic)
    return;

  // Enable local APIC; set spurious interrupt vector.
  lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));

  // The timer repeatedly counts down at bus frequency
  // from lapic[TICR] and then issues an interrupt.
  // If xv6 cared more about precise timekeeping,
  // TICR would be calibrated using an external time source.
  lapicw(TDCR, X1);
  lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
  lapicw(TICR, 10000000);      You, last month • Initial XV6

  // Disable logical interrupt lines.
  lapicw(LINT0, MASKED);
  lapicw(LINT1, MASKED);

  // Disable performance counter overflow interrupts
  // on machines that provide that interrupt entry.
  if(((lapic[VER]>>16) & 0xFF) >= 4)
    lapicw(PCINT, MASKED);
```

یکی دیگر از سخت افزار های دیگر که برای Timer راه اندازی می شود idinit می باشد
که timer interrupt را راه اندازی میکند.
یکی دیگر از وظایف آن راه اندازی جدول interrupt Desciptor یا همان IDT می باشد که ردیف های مربوط به interrupt های timer هم در آن ذخیره میشود.

```
void
idtinit(void)
801059c0: 55                   push   %ebp
{
  pd[0] = size-1;
801059c1: b8 ff 07 00 00       mov    $0x7ff,%eax
801059c6: 89 e5                mov    %esp,%ebp
801059c8: 83 ec 10             sub    $0x10,%esp
801059cb: 66 89 45 fa          mov    %ax,-0x6(%ebp)
  pd[1] = (uint)p;
801059cf: b8 c0 3c 11 80       mov    $0x80113cc0,%eax
801059d4: 66 89 45 fc          mov    %ax,-0x4(%ebp)
  pd[2] = (uint)p >> 16;
801059d8: c1 e8 10             shr    $0x10,%eax
801059db: 66 89 45 fe          mov    %ax,-0x2(%ebp)
  asm volatile("lidt (%0)" : : "r" (pd));
801059df: 8d 45 fa             lea    -0x6(%ebp),%eax
801059e2: 0f 01 18             lidtl  (%eax)
```

```
void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}
```

وقتی که timer ، یک interrupt ایجاد میکند این باید توسط interrupt handler مناسب پاسخ داده شود که وظیفه ی این بر عهده ی تابع tap در فایل trap.c می باشد.

```
void
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();
```

همانطور که نشان داده شده اگر مقدار trap سیستم برابر با مقدار بیس ادرس interrupt ها به علاوه ی آفست معادل T_IRQ0 + IRQ_TIMER timer باشد، پس یک timer interrupt رخ داده و وریبل ticks (واحد زمانی سیستم) یکی افزایش می یابد و تمامی فرایند های wait شده روی ticks یعنی فرایند هایی که برای event های زمان دار صبر کرده اند، شرط wait آن ها بررسی می شود تا در صورت ارضا شدن، به ادامه ی عملیات خود بپردازند.

سیستم درکی از زمان ندارد و برای کالیبره کردن زمان و شبیه سازی آن متغیر گلوبالی به نام ticks تعریف میکند و در هر واحد زمانی که طبق فرکانس صورت میگیرد(از وظایف lapicinit مشخص کردن همین فرکانس بود)آن را یک واحد افزایش میدهد، به این معنی که یک واحد زمانی گذشته است و پس از آن با اجرای دستور wakeup(&ticks) به پردازش تمامی time-based event ها می پردازد. (یعنی تمامی فرایند هایی که برای رخدادهای زمانی صبر کرده بودند بررسی میکند و در صورت برقراری شرط، آن ها را آزاد میکند)

```
acquire(&tickslock);
ticks++;
wakeup(&ticks);
release(&tickslock);
```

تنظیم LAPIC و تایمر

LAPIC، یا Local Advanced Programmable Interrupt Controller، در سیستم های چندپردازنده‌ای برای مدیریت وقفه ها برای هر CPU استفاده می‌شود. در xv6، LAPIC برای مدیریت وقفه های تایمر تنظیم می‌شود که برای زمان‌بندی کارها و نگهداری زمان بسیار مهم هستند.

```
lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
lapicw(TICR, 10000000);
```

این خط تایمر LAPIC را تنظیم می‌کند.
`lapicw`: این یک تابع است که برای نوشتن به یک رجیستر خاص LAPIC استفاده می‌شود.

```
void lapicw(int index, int value);
```

که در آن `index` مشخص می‌کند به کدام رجیستر LAPIC باید نوشت، و `value` داده‌ای است که باید نوشته شود.
TIMER: این شاخص رجیستر تایمر LAPIC است. مشخص می‌کند که ما قصد داریم تایمر را تنظیم کنیم.
PERIODIC: این یک پرچم است که تایمر را به حالت دوره ای تنظیم می‌کند. در حالت دوره ای، تایمر به طور مکرر و در فواصل منظم وقفه ها را ارسال می‌کند
(T_IRQ0 + IRQ_TIMER): این عبارت شماره وکتور برای وقفه تایمر را مشخص می‌کند. در اینجا:
T_IRQ0: این پایه شماره وکتور وقفه برای وقفه های سخت‌افزاری است.
IRQ_TIMER: این شماره درخواست وقفه خاص برای تایمر است.وقتی ترکیب می‌شود، `T_IRQ0 +` `IRQ_TIMER` شماره وکتور وقفه برای وقفه تایمر را تنظیم می‌کند.

این خط تایمر LAPIC را به حالت دوره ای تنظیم می‌کند و شماره وکتور وقفه برای وقفه تایمر را تنظیم می‌کند. این بدان معنی است که تایمر در فواصل منظم وقفه ها را تولید خواهد کرد که سیستم می‌تواند برای زمان‌بندی و سایر وظایف دوره ای از آنها استفاده کند.
خط دوم مقدار اولیه شمارش برای تایمر LAPIC را تنظیم می‌کند.
TICR: این شاخص رجیستر شمارش اولیه تایمر است. مشخص می‌کند که مقدار اولیه تایمر چه باشد.
10000000: این مقدار اولیه شمارش است که باید در تایمر بارگذاری شود.
این خط مقدار اولیه شمارش تایمر LAPIC را به 10000000 تنظیم می‌کند. وقتی تایمر شروع به کار می‌کند، از این مقدار به سمت صفر شمارش خواهد کرد. در حالت دوره‌ای، وقتی شمارش به صفر رسید، این مقدار اولیه دوباره بارگذاری می‌شود و شمارش مجدداً آغاز می‌شود.

نحوه ی ارزیابی برنامه با تست های مختلف:

نوشتن user program:

ابتدا در فایل c کد مورد نظر را مینویسیم،سپس در makefile در قسمت ها `UPROGS` و `EXTRA` فایل نوشته شده را اضافه میکنیم.

- **هدف از نوشتن برنامه هایی با زمان اجرای طولانی:**

برای اینکه تغییرات اعمال شده بهتر قابل درک باشد بهتر از برنامه هایی تست شوند که در یک دور فعالیت زمانبندی به اتمام نرسند و context switching بیشتری اتفاق بیفتد مانند برنامه هایی که swapping زیاد دارند(مانند ضرب ماتریس ها با ابعاد بزرگ) یا نیاز به I\O زیاد دارند(مانند disk I/O).
برنامه هایی که ایجاد کردیم را در شکل روبه رو میبینیم.

```
matmul          2 19 15892
diskwrite       2 20 15576
diskwriteL      2 21 18628
mdfork          2 22 20472
```

- **ضرب ماتریس ها:**

برای ضرب ماتریس ها user program ای نوشتیم که ۲ ماتریس نسبتا بزرگ را در هم ضرب میکند.برای تست کردن نیاز به زمان انجام فرایند قبل و بعد از تغییر را داریم.
ابتدا زمان اجرا را قبل از تغییر محاسبه میکنیم.
همانطور که مشخص است tick 9 زمان برده است.

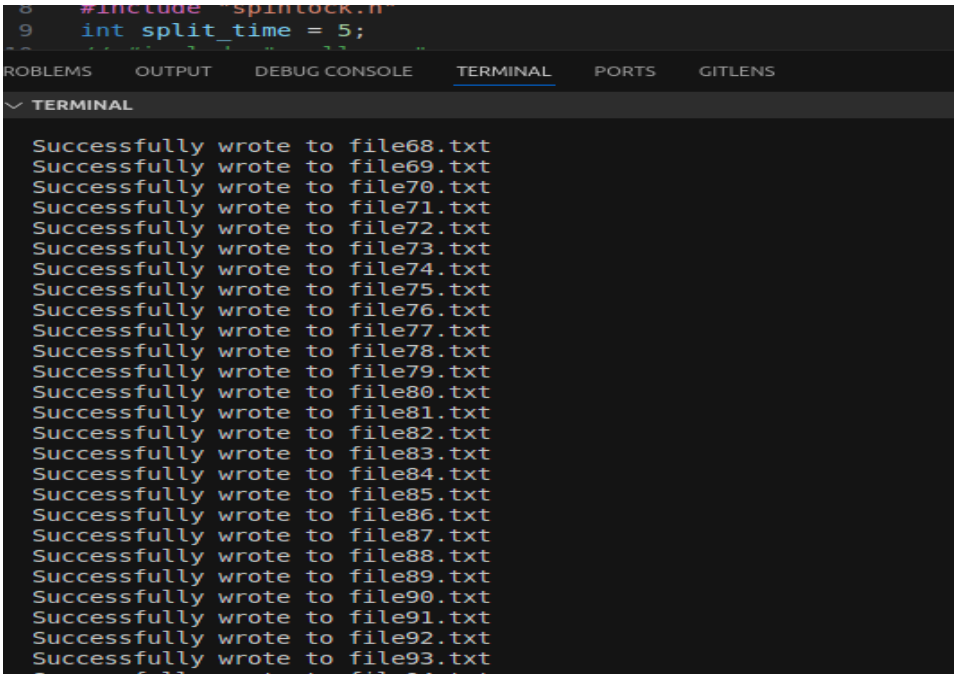
```
$ matmul
1240 1120 1000 880 760 640 520 400 280 160 40 -80 -200 -320 -440 -560
1360 1224 1088 952 816 680 544 408 272 136 0 -136 -272 -408 -544 -680
1480 1328 1176 1024 872 720 568 416 264 112 -40 -192 -344 -496 -648 -800
1600 1432 1264 1096 928 760 592 424 256 88 -80 -248 -416 -584 -752 -920
1720 1536 1352 1168 984 800 616 432 248 64 -120 -304 -488 -672 -856 -1040
1840 1640 1440 1240 1040 840 640 440 240 40 -160 -360 -560 -760 -960 -1160
1960 1744 1528 1312 1096 880 664 448 232 16 -200 -416 -632 -848 -1064 -1280
2080 1848 1616 1384 1152 920 688 456 224 -8 -240 -472 -704 -936 -1168 -1400
2200 1952 1704 1456 1208 960 712 464 216 -32 -280 -528 -776 -1024 -1272 -1520
2320 2056 1792 1528 1264 1000 736 472 208 -56 -320 -584 -848 -1112 -1376 -1640
2440 2160 1880 1600 1320 1040 760 480 200 -80 -360 -640 -920 -1200 -1480 -1760
2560 2264 1968 1672 1376 1080 784 488 192 -104 -400 -696 -992 -1288 -1584 -1880
2680 2368 2056 1744 1432 1120 808 496 184 -128 -440 -752 -1064 -1376 -1688 -2000
2800 2472 2144 1816 1488 1160 832 504 176 -152 -480 -808 -1136 -1464 -1792 -2120
2920 2576 2232 1888 1544 1200 856 512 168 -176 -520 -864 -1208 -1552 -1896 -2240
3040 2680 2320 1960 1600 1240 880 520 160 -200 -560 -920 -1280 -1640 -2000 -2360
Time taken: 9 ticks
process  (0)finished ------------------------------------------------------
turnaround time is for process matmul (pid = 6) is 1617
```

- **خواندن و نوشتن از دیسک:**

یک نمونه دیگر تست خواندن و نوشتن در فایل است.ابتدا یک user program نوشتیم که به صورت زیر اجرا می شود و در آن فایل متن مورد نظر را مینویسد.
در صورت نبود فایل،فایل را میسازد.

Usage: diskwrite <filename> <text>

```
Successfully wrote to file89.txt
Successfully wrote to file90.txt
Successfully wrote to file91.txt
Successfully wrote to file92.txt
Successfully wrote to file93.txt
Successfully wrote to file94.txt
Successfully wrote to file95.txt
Successfully wrote to file96.txt
Successfully wrote to file97.txt
Successfully wrote to file98.txt
Successfully wrote to file99.txt
Successfully wrote to file100.txt
Time taken: 61 ticks
process  (0)finished ------------------------------------
```



حال این دستور در زمان کوتاهی اجرا میشود،برای تست برنامه،ما به برنامه ای نیاز داریم که به زمان بیشتری نیاز داشته باشد.بنابراین یک user program دیگه ای نوشتیم که با گرفتن متنی،آن را در ۱۰۰ فایل مینویسد.(ایندکس شروع اسم فایل ها نیز در ورودی گرفته میشود)

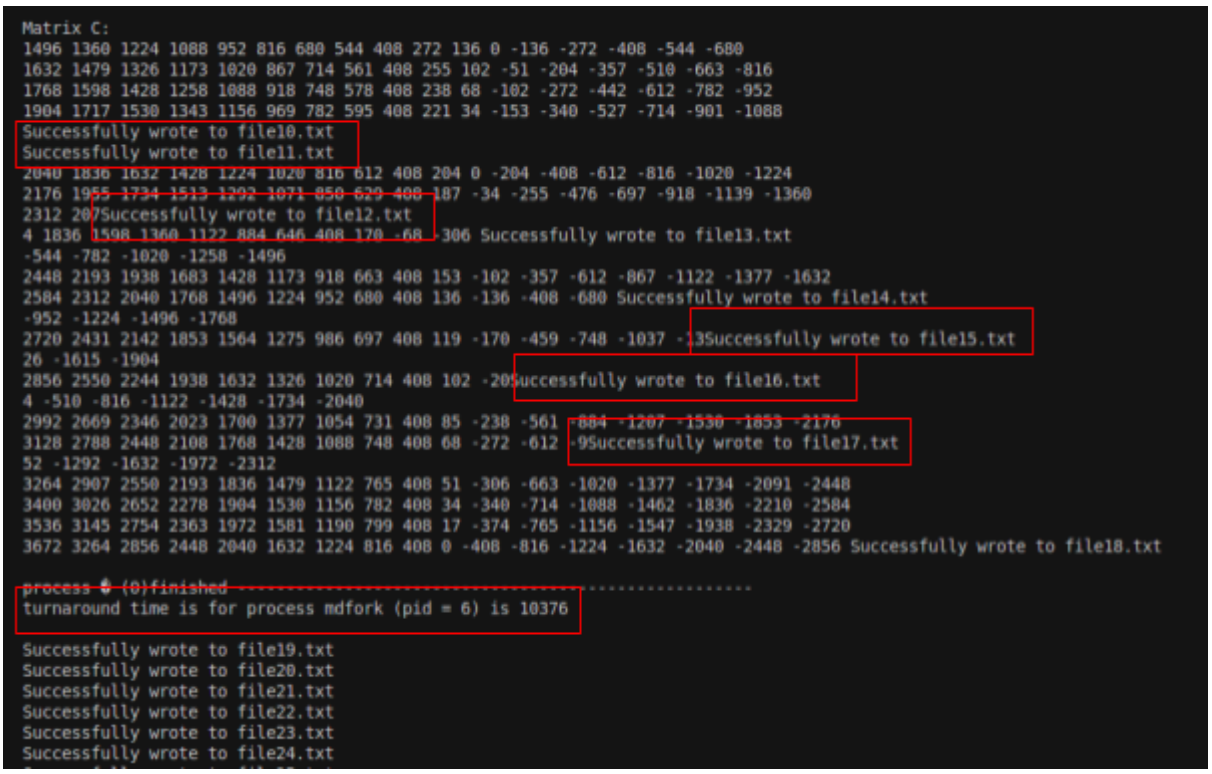
ورودی: diskwriteL "hello" 0

خروجی: تصویر بالا

همانطور که قابل مشاهده است tick 97 زمان برده است.

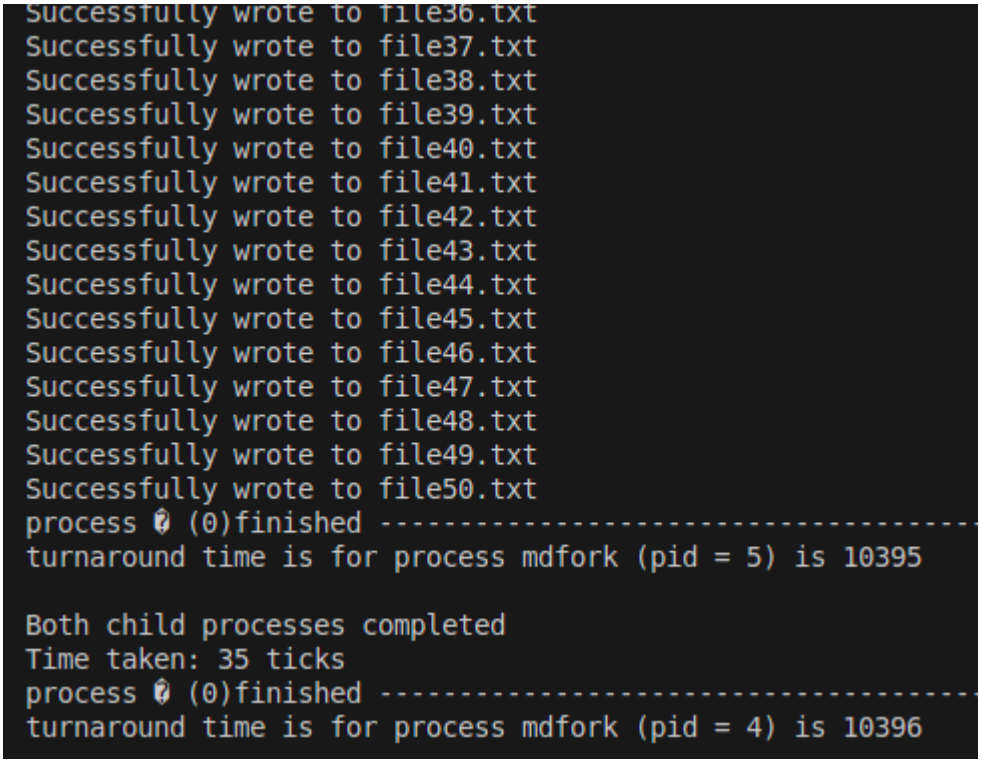
- **تست با استفاده از ترکیت ۲ فایل قبلی و fork:**

حال یک user program دیگه ای نوشتیم که دو مورد قبل را با استفاده از فورک هم زمان اجرا میکند.به این صورت که یک فرزند ضرب ماتریس ها را انجام میدهد و فرزند تولید شده ی دیگر ۱۰۰ فایل را تولید و در ان ها مینویسد.



همانطور که در شکل پیداست در هنگام ضرب ماتریس ها، فایلهایی نیز نوشته و ذخیره شده اند که نشان از context switch میان فرآیند های فرزند و والد می باشد

و بعد از اتمام محاسبه ی ماتریس ها و terminate شدن فرآیند مربوط به آن، فرآیند دیگر به کار خود بدون context switch ادامه میدهد و تمام فایل ها را مینویسد و ذخیره میکند.



همانطور که از خروجی ها بر می آید نتیجه میگیریم که در هنگام فورک برنامه دو فرآیند با نام های mdfork و آیدی های 5 و 6 ایجاد میشوند که آیدی 6 مسئولیت محاسبه ی ماتریس را دارد که پس از اتمام محاسبه زودتر terminate میشود و در حین محاسبه با mdfork با آیدی 5 context switch میشود و پس از اتمام آن، 5 mdfork تمام فایل ها را بدون context switch با فرآیند دیگر، می نویسد و ذخیره میکند و در انتها خودش terminate میشود.

- **تغییر کوانتوم:**

در فایل proc.h:
ابتدا یک فیلد جدید time_slice برای هر فرآیند اضافه میکنیم به این دلیل که هر فرآیند به اندازه کوانتوم مورد نظر اجرا شود،در واقع کنترل کردن زمان اجرای هر فرآیند در یک دور فعالیت زمانبندی است.

```
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];
  int time_slice;              //
};
```

در فایل trap.c:
حال باید time_slice داده شده به هر فرآیند را که م
تعیین شده ای است برای کنترل کوانتومی از زمان
که هر فرآیند می تواند مصرف کند تا وضعیت
عدالتمندی در سیستم داشته باشیم.
در هر timer interrupt یک واحد این
timer_slice مربوط به هر فرآیند را افزایش
میدهیم.
این کار را تا وقتی انجام میدهیم تا مقدار
time_slice برابر مقدار متغیر گلوبال
time_slice شود.
وقتی شد مقدار آن را ابتدا به مقدار صفر تغییر
میدهیم و بعد از آن با استفاده از دستور ، cpu
yield را از فرآیند میگیریم تا فرآیند بعدی cpu را به دست آورد و context switch اتفاق بیفتد.

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  if(myproc() && myproc()->state == RUNNING) {
    myproc()->time_slice++;
    if(myproc()->time_slice >= time_slice){
      myproc()->time_slice = 0;
      yield();
    }
  }
```

```
static struct proc *initproc;
int time_slice = DEFAULT_TIME_SLICE;
int nextpid = 1;
```

مقدار slice_time که به صورت global تعریف میکنیم تا همه ی فرآیند ها به آن دسترسی داشته باشند و یا آن را بتوانند تغییر دهند.

```
#define FSSIZE       1000  // size
#define DEFAULT_TIME_SLICE 3
```

برای slice_time مقدار دیفالت تعیین می کنیم که کوانتوم الگوریتم round robin است.

```
uint ticks;
extern int time_slice;
```

به دلیل نیاز به time_slice در فایل trap.c نیاز است که این متغیر را به صورت extern تعریف کنیم تا در فایل های دیگر از جمله trap.c دسترسی داشته باشیم و بتوانیم تغییرات آن را بفهمیم.

```
p->state = EMBRYO;
p->init_time = ticks;
p->pid = nextpid++;
p->time_slice = 0;
p->terminate_time = 0;
release(&ptable.lock);
```

Initial کردن مربوطه در هنگام initial شدن یک فرآیند

```
// Jump into the scheduler, never to return.
curproc->state = ZOMBIE;
curproc->terminate_time = ticks;
curproc->init_time = 0;
cprintf("process %s (%d)finished ----------------------------------------------------\n",p->name, p->pid);
sched();
panic("zombie exit");
}
```

ریست کردن init_time در هنگام termination یک فرآیند و نمایش زمان سپری شده برای آن از هنگام initial شدن در سیستم تا termination که همان turnaround time می باشد.

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

متغیر terminated را به عنوان فلگی برای اطلاع از terminate شدن فرآیندی در یک دور پیمایش کامل فرآیند ها، تعریف میکنیم.

```
acquire(&ptable.lock);
terminated = 0;
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;
```

مقدار این فلگ بعد از هرپیمایش و قبل از شروع پیمایش جدید همه ی فرآیند ها ریست میشود.

```
swtch(&(c->scheduler), p->context);
if (p->terminate_time >0 & terminated == 0)      You, 1 second ago • Uncommitted changes
{
  terminated += 1;
  cprintf("turnaround time is for process %s (pid = %d) is %d\n\n",p->name, p->pid, (p->terminate_time - p->init_time));
}
switchkvm();
```

بعد از دستور swtch از cpu به فرآیند، زمانی به scheduler برمیگردیم که فرآیند cpu را yield کرده باشد، یا به خاطر مصرف کوانتوم زمانیش یا بخاطر درخواست I/O یا به صورت اختیاری، همانطور که در تابع exit دیدیم، در هنگام terminate شدن هر فرآیند، مقدار terminate_time آن ثبت میشود پس حتما فرآیندی که terminate شده، مقدار terminate time آن مقداری بزرگ تر از صفر (مقدار اولیه) دارد همچنین با چک کردن فلگ terminated وارد شرط if میشویم(زیرا حتی یک terimnate شدن فرآیند برای دوبرابر نکردن تایم اسلایس برای حلقه ی بعد کافی است)
و در حلقه مقدار terminated را یکی افزایش میدهیم و turnaround time را نمایش می دهیم.

```
}
if (terminated == 0)
{
    if (time_slice > 1000000)
    {
      time_slice = 2;
    }
    else{
      time_slice *= 2;
    }

}

release(&ptable.lock);
```

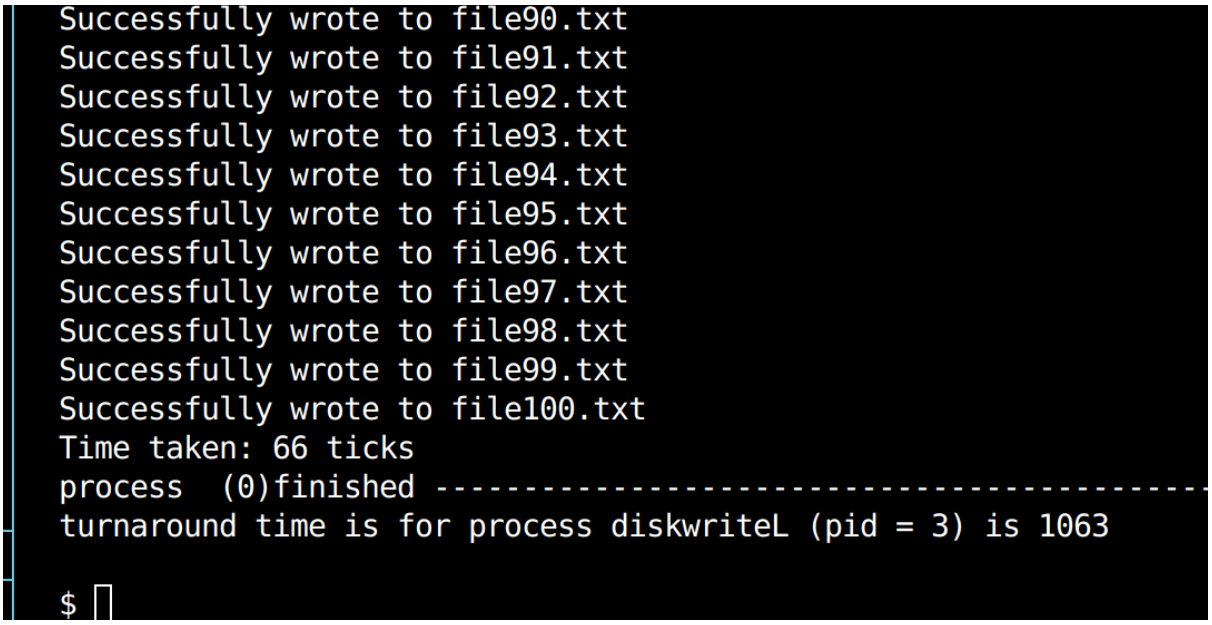
در آخر بعد از اتمام پیمایش همه ی فرآیندها، در صورتی که فلگ terminated مقدار اولیه ی خود را داشت و هیچ فرآیندی در طول پیمایش، به پایان نرسیده بود، مقدار حد بالای کوانتوم زمانی برای همه فرآیندها دوبرابر میشود و فرآیند ها برای پیمایش بعدی میتوانند دو برابر زمان قبلی خود، از cpu استفاده کنند تا قبل از اینکه قبضه شوند.
در اینجا به دلیل تعداد کمتر process ها، فرآیندهای کمتری در واحد زمان terminate میشوند پس این پیمایش تمام فرآیندها در واحد زمان، به تعداد بیشتری بدون terminate شدن هیچ فرآیندی تمام میشود و در نتیجه time_slice با سرعت خیلی بیشتری دوبرابر میشود، پس برای overflow رخ ندادن یک حد بالا (یک میلیون) برای time_slice درنظر گرفتیم و در صورت تجاوز از آن، مقدار آن را به 2 ریست میکنیم.

- پس از تغییر دادن الگوریتم حال دوباره برنامه هایی که برای ارزیابی نوشته بودیم را اجرا میکنیم.
- ضرب ماتریس ها

```
$ matmul
1240 1120 1000 880 760 640 520 400 280 160 40 -80 -200 -320 -440 -560
1360 1224 1088 952 816 680 544 408 272 136 0 -136 -272 -408 -544 -680
1480 1328 1176 1024 872 720 568 416 264 112 -40 -192 -344 -496 -648 -800
1600 1432 1264 1096 928 760 592 424 256 88 -80 -248 -416 -584 -752 -920
1720 1536 1352 1168 984 800 616 432 248 64 -120 -304 -488 -672 -856 -1040
1840 1640 1440 1240 1040 840 640 440 240 40 -160 -360 -560 -760 -960 -1160
1960 1744 1528 1312 1096 880 664 448 232 16 -200 -416 -632 -848 -1064 -1280
2080 1848 1616 1384 1152 920 688 456 224 -8 -240 -472 -704 -936 -1168 -1400
2200 1952 1704 1456 1208 960 712 464 216 -32 -280 -528 -776 -1024 -1272 -1520
2320 2056 1792 1528 1264 1000 736 472 208 -56 -320 -584 -848 -1112 -1376 -1640
2440 2160 1880 1600 1320 1040 760 480 200 -80 -360 -640 -920 -1200 -1480 -1760
2560 2264 1968 1672 1376 1080 784 488 192 -104 -400 -696 -992 -1288 -1584 -1880
```

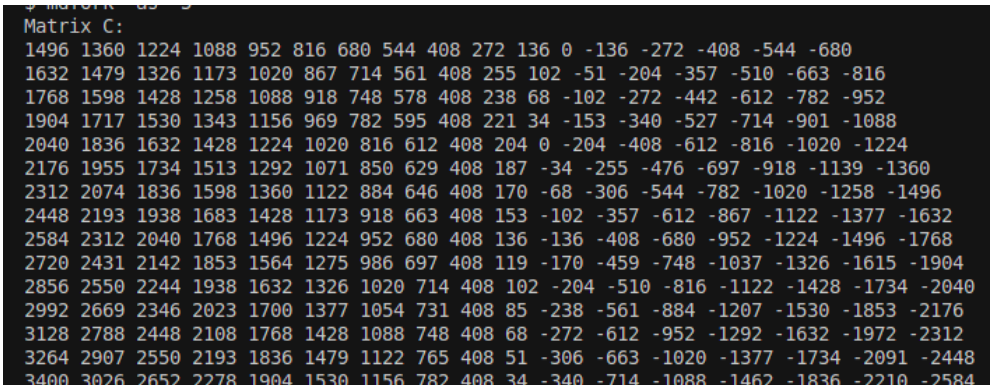
در ضرب ماتریس ها می بینیم که turn around time افزایش یافته.میتوان دریافت کرد که با افزایش time_slice همیشه turn around time کاهش نمی یابد.

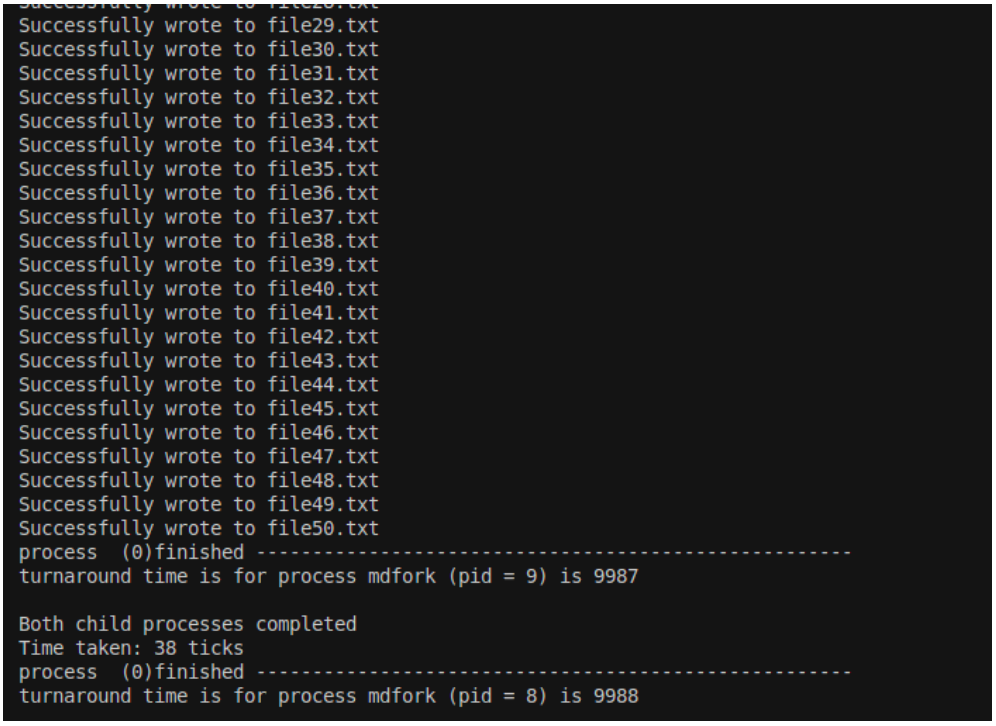
- نوشتن در دیسک



همانطور که در تصویر بالا میبینید turn around time نسبت به الگوریتم قبل کاهش یافته است.به دلیل اینکه با افزایش کوانتوم context switching کمتری داریم پس در نتیجه turn around time کمتری داریم.

- استفاده از fork





همانطور که از عکسای mdfork که بیش از یک فرآیند همزمان درگیر انجام تسک می باشند، تاثیر بیشتری به چشم می آید،
این الگوریتم در نهایت سیری به سمت FIFO شدن دارد و پس از هر پیمایش ناموفق(بدون کامل شدن حتی یک فرآیند) با افزایش کوانتوم زمانی، الگوریتم را بیشترو و بیشتر به FIFO نزدیک میکند و همانطور که از تصویر بر می آید در ابتدا mdfork با آیدی 10 بدون context switch (به دلیل زیاد بدون time_slicce) ماتریکس را محاسبه میکند و ترمینیت میشود و بعد از آن mdfork با آیدی 9 به I/O های خود میپردازد و تمامی فایل ها را writeو save میکند، بدون مداخله و context switch با فرآیند دیگری پس در این مثال و وضعیت خاص، این الگوریتم با کاهش turaround time و افزایش performance همراه خواهد بود.