گزارش فاز سوم پروژه xv6

1/26/2022

محمد جواد زندیه، محمد جواد رجبی
دانشگاه صنعتی امیر کبیر

سوالات اولیه برای شروع پروژه:

1. خط مشی به صورت پیش فرض چه پردازه ای را برای اجرا انتخاب می کند؟

در فایل main.c که نقطه شروع سیستم عامل می باشد، تابع ()userinit از فایل proc.c صدا زده میشود تا اولین پردازه کاربر ساخته شود.

فایل main.c :

```c
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  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
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

فایل proc.c و تابع ()userinit :

در این فایل برای یک پردازه که در ابتدا null است مقادیر را ست می کنیم از جمله page directory و ...
و در انتها هم حالت آن را به حالت Runnable تغییر می دهیم تا به حالت اجرا در بیاید و این اولین پردازه ای است که اجرای خود را شروع می کند.

```c
//PAGEBREAK: 32
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  p->ctime = ticks;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;

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

در انتها وقتی که cpu برای اولین بار تابع scheduler را از فایل proc.c صدا می زند، آن پردازه ابتدایی که ساخته شده بود بر گردانده می شود(چون در حالت Runnable بود و تنها پردازه موجود هم بود)
تابع scheduler یک پردازه را بر می گرداند که cpu آنرا اجرا کند.

2. وقتی که یک پردازه از رخداد IO باز می گردد چه اتفاق می افتد؟

وقتی که wakeup صدا زده می شود تمام پردازه هایی که در حالت sleeping بوده اند به حالت running می روند و state شان تغییر خواهد کرد و دوباره به صف انتظار برای دریافت cpu باز می گردند.

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

```c
//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan){
      p->state = RUNNABLE;
      if (getP() == 3) //DM1
      {
        p->priority = 1;
      }

    }
}
```

3. وقتی که یک پردازه ایجاد می شود، چه اتفاق می افتد و زمان بندی در چه زمان هایی و با چه فاصله زمانی انجام می شود؟

هر پردازه بعد از ایجاد باید به ptable اضافه گردد تا بتوان در scheduler زمانی که به مدیریت پردازه ها می پردازیم این پردازه را هم مدیریت کنیم.

مدیریت زمان بندی در scheduler که از توابع proc.c می باشد انجام می شود و هر گاه که پردازه cpu را آزاد کرد یا آنکه بر اساس trap ای که در فایل trap.c پیاده سازی شده است، cpu از آن گرفته شد حال باید در scheduler روی تمام پردازه ها iterate کنیم تا پردازه ای که آماده اجرا است و سیاست را بر آورده می کند پیدا کرده و با یک context switch ز سی پی یو را به آن بدهیم (یا درواقع پردازه cpu را برابر با این پردازه کنیم😊)

به طور پیش فرض از سیاست round robin در xv6 استفاده می شود و در هر clock tick سی پی یو از پردازه گرفته می شود(یا ایجاد trap که در فایل trap.c پیاده سازی شده است)

تصویر scheduler:

به کامنت ها که نحوه کار scheduler را بیان می کند توجه کنید.(همان توضیحات بالا می باشد)

```
//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p = 0;
  struct cpu *c = mycpu();
  c->proc = 0;

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

تصویری از فایل trap.c در جایی که به صورت پیش فرض در هر clock tick ترپ داده می شود و yield صورت می گیرد تا عملیات پس گرفتن cpu صورت گیرد.

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

در yield سی پی برای پیدا کردن پردازه جدید به scheduler مراجعه می کند.

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

تابع sched تابعی است که پیش از ورود به scheduler صدا زده می شود و عملیات context switch در آن انجام می شود.

```c
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state. Saves and restores
// intena because intena is a property of this
// kernel thread, not this CPU. It should
// be proc->intena and proc->ncli, but that would
// break in the few places where a lock is held but
// there's no process.
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

توضیح فایل های تغییر داده شده در پروژه :

بخش اول: پیاده سازی الگوریتم های زمان بندی:

1. الگوریتم round robin در xv6

ابتدا در فایل proc.h و در struct proc یک شمارنده به نام tickcounter تعریف می کنیم. این شمارنده تعداد clock tick های اجرای یک پردازه را نگهداری می کند.

```c
  int tickcounter;
```

سپس یک سیستم کال به نام inctickcounter در proc.c تعریف می کنیم که وظیفه افزایش یک واحدی tickcounter را دارد و در هر کلاک نیاز داریم که برای هر پردازه آنرا افزایش دهیم.

```c
int inctickcounter() {
  int res;
  acquire(&ptable.lock);
  res = ++myproc()->tickcounter;
  release(&ptable.lock);
  return res;
}
```

در فایل param.h هم متغیری به نام QUANTUM با مقدار اولیه 10 define می کنیم و در واقع همان time quantum برای الگوریتم round robin می باشد.

```c
#define QUANTUM       10 // quantum for round-robin scheduling policy
```

سپس در فایل trap.c در جایی که قرار است trap ای برای گرفتن cpu از پردازه بشود، آیا پردازه مورد نظر به اندازه کوانتوم زمانی اجرا شده است یا نه و اگر اجرا نشده بود به اندازه یک کوانتوم، پردازه از آن گرفته نمی شود.

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

دقت شود که در trap.c باید سیستم کال inctickcounter را هم extern کنیم تا به آن دسترسی داشته باشیم.

```c
extern int inctickcounter(void);
```

2. الگوریتم زمان بندی صف اولویت (Non-preemptive priority scheduling):

در struct proc در فایل proc.h باید یک فیلد دیگر به نام priority اضافه کنیم که اولویت پردازه را برایمان نگهدارد.

به صورت پیش فرض اولویت پردازه ها برابر با 3 می باشد و این را زمانی که یک پردازه ایجاد می شود باید برای آن ست کنیم. یعنی در تابع allocproc(void) از فایل proc.c و در قسمت found :

```c
  p->priority = 3;     // default priority
```

به یک سیستم کال هم به نام setPriority نیاز داریم که با گرفتن یک pid و priority اولویت پردازه را به اولویت مورد نظر تغییر می دهد.

```c
int
sys_setPriority(void)
{
  int pid , priority;
  if (argint(0, &pid) < 0)
  {
    return -1;
  }
  if (argint(1, &priority) < 0)
  {
    return -1;
  }

  return setpri(pid , priority);
}
```

```c
// change priority
int setpri( int pid, int priority)
{

  struct proc *p;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      if(priority >= 1 && priority <= 6)
        p->priority = priority;
      else
        p->priority = 5;
      break;
    }

  }

  release(&ptable.lock);
```

همچنین در تابع scheduler نیاز داریم تا از ptable پردازه با بیشترین اولویت را در هر زمانی که نیاز به پیدا کردن پردازه جدید برای اجرا شد بدست آوریم.

```
struct proc *p1;
struct proc *highp = 0;

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

  highp = p;

  for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
    if(p1->state != RUNNABLE)
      continue;
    if (highp->priority > p1->priority)
    {
      highp = p1;
    }
  }
  p = highp;
```

3. <u>الگوریتم زمان بندی طبق صف چند لایه(به شکل preemptive)</u>

در این سیاست از دو سیاست قبل استفاده می کنیم و SML را پیاده سازی می کنیم.

در تابع scheduler با صدا زدن تابع findReadyProcess پردازه بعدی برای اجرا طبق سیاست SML را پیدا می کنیم.

در findReadyProcess 6 عدد index داریم که نمایان گر ایندکس فعلی مان در هر یک از 6 صف می باشد و فرایند پیدا کردن پردازه هر صف به صورت زیر است:

```
proc2 = &ptable.proc[(*index1 + i) % NPROC];
if (proc2->state == RUNNABLE && proc2->priority == *priority) {
  *index1 = (*index1 + 1 + i) % NPROC;
  return proc2; // found a runnable process with appropriate priority
}
```

مابقی توضیحات به صورت کامنت در این تابع قرار داده شده است.

در فایل trap.c نیز چون باید صف ها از round robin پیروی کنند و طبق شماره صف اولویت آنها افزایش یابد پس باید به این صورت در هر صف trap ایجاد کنیم برای پردازه های آن صف:

```
if (myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0 + IRQ_TIMER && inctickcounter() == (7 - getPri()) * QUANTUM)
  yield();
```

4. الگوریتم زمان بندی صف چند لایه پویا:

این سیاسیت همانند سیاست قبل است فقط باید قوانینی که خواسته شده است را اجرا کنیم:

۱) فراخوانی سیستم کال exec اولویت پردازه را به حالت پیش‌فرض ریست می‌کند (default priority).

در فایل exec.c باید اولویت پیش فرض را روی 3 قرار دهیم.

```
if (policy == 3) //DML
{
  curproc->priority = 3;
}
```

۲) برگشت از حالت sleep mode یا همان IO باعث افزایش اولویت پردازه به بیشترین اولویت می شود یا همان عدد ۱ می‌شود (highest priority).

```
//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan){
      p->state = RUNNABLE;
      if (getP() == 3) //DM1
      {
        p->priority = 1;
      }

    }
}
```

۳) واگذاری (Yielding) cpu به صورت دستی اولویت پردازه را تغییر **نمی‌دهد**.

به صورت پیش فرض این چنین است.

۴) اجرا شدن به اندازه ی یک full quanta باعث کاهش اولویت پردازه به اندازه ۱ واحد می‌شود.

پس از هر بار که پردازه اجرا شد و با ترپ خواستیم cpu را از آن بگیریم باید از تابع decpriority که در proc.c نوشته ایم استفاده کنیم تا اولویت را یک واحد تغییر دهیم.

```
void decpriority(void) {
  // acquire(&ptable.lock);
  myproc()->priority = myproc()->priority  == 1 ? 1 : myproc()->priority  - 1;
  // release(&ptable.lock);
}
```

در فایل trap.c:

```
if (myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0 + IRQ_TIMER && inctickcounter() == (7 - getPri()) * QUANTUM){

      decpriority();
      yield();
}
```

بخش دوم: کد های کمکی یا ارزیابی پیاده سازی های انجام شده:

1. changePolicy:

در این بخش باید بتوانیم سیاست را تغییر دهیم. ابتدا یک فلگ به نام policyy در proc.c تعریف می کنیم و سپس آنرا به صورت پیش فرض برابر با 0 یعنی همان round robin قرار می دهیم و توابع زیر آنرا ست یا آنرا دریافت می کنیم.

```
int setP(int policy){

  policyy = policy;

  return 0;
}

int getP(void){

  return policyy;
}
```

در تمام بخش های کد باید چک شود که در کدام سیاست هستیم و تکه کد مربوط به آن سیاست را انجام دهیم.

2. قابلیت اندازه گیری زمان:

در proc.h و در بلوک مربوط به پردازه باید متغیر های جدیدی را تعریف کنیم و در proc.c هم مقدار دهی اولیه کنیم آنها را و هم اینکه در مواقع لزوم آن ها را تغییر دهیم

```c
uint ctime;                    // Process creation time
uint ttime;                    // Process termination time
int stime;                     //process SLEEPING time
int retime;                    //process READY(RUNNABLE) time
int rutime;                    //process RUNNING time
```

مقدار دهی اولیه: در تابع allocproc از proc.c

```c
p->ctime = ticks;
p->ttime = 0;
p->retime = 0;
p->rutime = 0;
p->stime = 0;
```

و همچنین در تابع wait از proc.c

```c
for(;;){
  // Scan through table looking for exited children.
  havekids = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent != curproc)
      continue;
    havekids = 1;
    if(p->state == ZOMBIE){
      // Found one.
      pid = p->pid;
      kfree(p->kstack);
      p->kstack = 0;
      freevm(p->pgdir);
      p->pid = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->ctime = 0;
      p->ttime = 0;
      p->state = UNUSED;
      release(&ptable.lock);
      return pid;
    }
  }
```

در تابع newwait هم همین مقدار دهی را انجام می دهیم

یک تابع به نام updatetimes در proc.c ایجاد کرده ایم که در هر clock tick مقادیر زمان ها را آپدیت کنیم.

```c
//This method will run every clock tick and update the statistic fields for each pro
void updatetimes() {
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    switch(p->state) {
      case SLEEPING:
        p->stime++;
        break;
      case RUNNABLE:
        p->retime++;
        break;
      case RUNNING:
        p->rutime++;
        break;
      default:
        ;
    }
  }
  release(&ptable.lock);
}
```

در انتها هم تابعی به نام getPerformance داریم که اطلاعات cpu burst time و waiting time و turnaround time یک پردازه را که بخواهیم از روی تایم های بالا که برای هر کدام بدست آورده بودیم پیدا می کند:

```c
int getPerformance(int pid, int *Btime, int *Wtime, int *TAtime){
  *Btime = 0; // cpu burst time
  *Wtime = 0; // waiting time
  *TAtime = 0; // turnaround time

  struct proc *p;

  acquire(&ptable.lock);

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->pid == pid){
      *TAtime = p->stime + p->retime + p->rutime;
      *Btime = p->rutime;
      *Wtime = p->stime + p->retime ;
      break;
    }

  }
  // cprintf("in proc.c  : tatime = %d , btime = %d , wtime = %d \
  release(&ptable.lock);

  return 0;
}
```

بخش سوم: تست نویسی

1. roundRobinTest:

همان طور که گفته شده است با استفاده از 10 fork فرزند برای پردازه پدر می سازیم و با انجام یک لوپ یک زمانی را به صورت بی خود سپری می کنیم و در نهایت موارد زیر را برای هر پردازه استخراج می کنیم و در آخر هم میانگین این تایم ها را با توجه به تعداد پردازه ها استخراج می کنیم:(CBT و Waiting Time، Turn Around Time)

برای این کار نیاز داریم تا این زمان ها توسط پدر پردازه ها استخراج شود پس وقتی پردازه ای به حالت zombie در آمد باید پدر اطلاعات را استخراج کند پس نیاز به یک تابع isZombie در proc.c نیز می باشیم:

```c
// check is it in Zombie state or not
int isZombie(int pid)
{

  struct proc *p;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      if (p->state == ZOMBIE)
      {
        release(&ptable.lock);
        return 1;
      }else
      {
        release(&ptable.lock);
        return 0;
      }


      break;
    }

  }
```

تست مربوط به این بخش در فایل roundRobinTest.c می باشد.

با کوانتوم 10

چک کردن خروجی با کم و زیاد کردن کوانتوم:

با کوانتوم 5

با کوانتوم 15

همانطور که مشاهده می شود با کاهش کوانتوم، مدت زمان انتظار یعنی Wtime کاهش پیدا کرده است.

2. prioritySchedTest:

تست مربوط به این بخش در فایل priorytest.c می باشد.

3. multiLayeredQueuedTest:

تست مربوط به این بخش در فایل SMLtest.c می باشد.

4. DynamicMultiLayeredQueuedTest:

تست مربوط به این بخش در فایل DMLtest.c می باشد.