به نام خدا

گزارش سوم آزمایشگاه سیستم عامل

سید علیرضا میرشفیعی - ۸۱۰۱۰۱۵۳۲
امیر حسین صفری - ۸۱۰۱۰۱۵۷۱
محمد صدرا عباسی - ۸۱۰۱۰۱۴۶۹

(1

ساختار PCB:

```
// 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)
```

ساختار PCB درون بخش proc.h

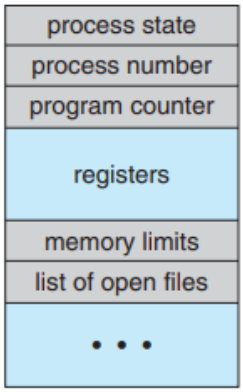


Figure 3.3 Process control block (PCB).

- enum procstate state : Process state در هر دو ساختار وجود دارند که به وضعیت پردازه اشاره دارند
- int pid : Process number که شناسه پردازه میباشد.
- struct trapframe *tf->eip : Program counter درون استراکت trapframe فیلد eip همان program counter هست که ادرس دستور بعدی می باشد.
- char *kstack , pde_t* pgdir , uint sz : Registers که sz برابر اندازه حافظه ، pgdir همان page directory و kstack ادرس ابتدای kernel stack مربوط به پردازه است.
- struct file *ofile : List of open files آرایه ای از اشاره گر ها به فایل هایست که پردازه باز کرده.

وضعیت های پردازه:

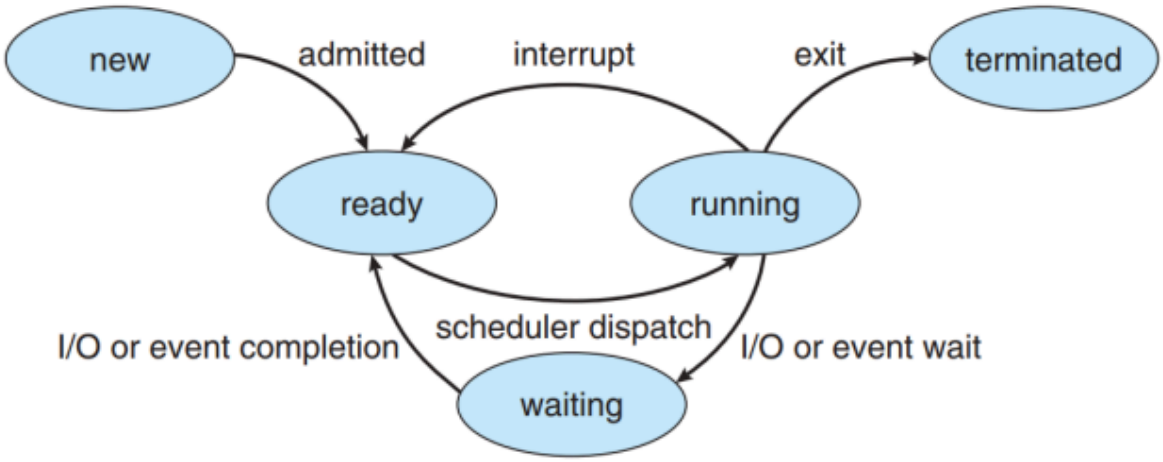
```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

- UNUSED: پردازه مورد استفاده قرار نگرفته
- EMBRYO: پردازه به تازگی در حال ایجاد شدن هست
- SLEEPING: پردازه منتظر یک event است
- RUNNABLE: آماده اجرا اما هنوز اجرا نشده
- RUNNING: در حال اجرا روی پردازنده
- ZOMBIE: پردازه پایان یافته اما منابعش هنوز آزاد نشده

(2


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


وضعیت پردازه ها



شکل ۱. چرخهٔ وضعیت یک پردازه.

- UNUSED : (معادل ندارد)
- EMBRYO : (new)
- SLEEPING : (waiting)
- RUNNABLE : (ready)
- RUNNING : (running)
- ZOMBIE : (terminated)

(3

درون proc.c زمانی که تابع userinit() اجرا میشود این اتفاق می افتد:
درون این تابع اول یک پردازه با استفاده از allocproc() گرفته میشود و درون این تابع state آن برابر EMBRYO که همان new میباشد ست میشود، سپس پس از گرفتن مقادیر اولیه درون تابع userinit() حالت RUNNABLE برای این تابع ست میشود که برابر همان حالت ready میباشد.

(4

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

Ptable in proc.c

مقدار NPROC برابر ماکز تعداد پردازه های ممکن در xv6 میباشد.
که در param.h این مقدار برابر 64 ست شده.

زمانی که تعداد پردازه ها از مقدار NPROC عبور کند، در proc.c زمانی که تابع allocproc برای fork یک پردازه جدید فراخوانده میشود، سطح کرنل صرفا این اجازه داده نمیشود و allocproc شکست میخورد.
در نتیجه این شکست در سمت کاربر بسته به کد، مقدار -1 برگردانده میشود، که همان نشانه شکست fork میباشد.

(5

از آنجایی که ptable یا همان جدول پردازه ها، یک ساختار مشترک میان همه پردازنده ها دارد، پس اگر دو پردازنده بصورت همزمان در حال اعمال تغییر بر روی این جدول باشند ممکن است هردو بر روی یک پردازه مشترک عملیات انجام دهند. که این اتفاق سبب ثبت داده های نادرست در پردازه میشود.

در صورتی که سیستم تک پردازه باشد، درست است که دیگر پردازنده دیگری وجود ندارد که بطور مستقیم در پردازه ها تغییر ایجاد دهد، اما همچنان ممکن است یک پردازه در پردازه دیگر تغییری ایجاد دهد. که مانند حالت قبل باعث ثبت داده های نادرست شود.

(6

این موضوع کاملا بستگی به مکان آن پردازه در ptable.proc و همچنین مکان ایندکس scheduler میباشد
اگر مکان پردازه در ptable.proc بعد از مکان ایندکس scheduler باشد در همان iteration پردازه اجرا خواهد شد.
اما اگر مکان پردازه در ptable.proc قبل از مکان ایندکس scheduler باشد، در iteration بعدی scheduler از این پردازه گذر میکند تا آن را اجرا کند.

(7

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

Struct context

- edi - Extended Destination Index: به عنوان مقصد استفاده میشود
- esi - Extended Source Index: به عنوان مبدا داده استفاده میشود
- ebx - Base Register: یک رجیستر چند منظوره که در نگهداری داده ها، اشاره گرها و... ممکنه استفاده بشه
- ebp - Base Pointer: نشان دهنده ابتدای stack frame
- eip - Extended Instruction Pointer: همان program counter میباشد.

(8

eip برابر همان program counter است که با فراخوانی کد اسمبلی switch.S ذخیره میشود

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

Switch.S

زمانی که تابع switch فراخوانی میشود ذخیره سازی طبق کد اتفاق می افتد.

(9

اگر تابع sti در اول هر iteration فعال نشود، به عبارتی دیگر وقفه ها فراخوانی نمیشوند پس در نتیجه دیگر تابع yield نیز فراخوانی نخواهد شد، به همین سبب پردازه ها حتی پس از اتمام زمانشان، منابعشان آزاد نمیشوند و بدون وقفه به عملیات ادامه میدهند. در نتیجه سیستم بدون در نظر گرفتن وقفه ها غیر قابل مدیریت میشود و در یک حلقه بی نهایت در یک پردازه loop میشود.

(10

```
// 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);
```

Code in lapic.c

از این قالب کد میتوان مقدار اولیه تایمر که برابر 10,000,000 و همچنین مقدار تقسیم کننده آن که برابر یک کلاک هست را مشاهده کرد.
در نتیجه بسته بر اینکه کلاک با چه فرکانسی باشد این مقدار میتواند متغیر باشد.
درصورتی که فرض کنیم مقدار کلاک برابر 1GHz باشد، تایمر 10 میلی ثانیه خواهد بود.
یعنی در هر 10 میلی ثانیه یک وقفه ارسال میشود.

(11

همانطور که در سوال 9 گفته شد، تابع yield در پاسخ interrupt صدا زده میشود.

```
void yield(void) {
  acquire(&ptable.lock);        // قفل جدول پردازه‌ها
  myproc()->state = RUNNABLE; // تغییر وضعیت از RUNNING به RUNNABLE
  sched();                      // رفتن به زمان‌بند برای انتخاب پردازه جدید
  release(&ptable.lock);        // آزادسازی قفل
}
```

yield definition

همانطور که در کد مشاهده میشود در این مرحله پردازه به حالت RUNNABLE در می آید و پس از فراخوانی sched تابع switch درون آن صدا زده میشود و switch conext اتفاق می افتد.

(12

از قبل میدانیم که 10ms فاصله بین هر وقفه هست، xv6 بصورت خودکار به هر پردازه یک واحد کوانتومی فرصت میدهد که در نتیجه برابر همان 10ms برای هر پردازه خواهد بود.

(13

```
  // Wait for children to exit.  (See wakeup1 call in proc_exit.)
  sleep(curproc, &ptable.lock);  //DOC: wait-sleep
}
```

code from wait()

همانطور که مشاهده میشود، در پایان حلقه بینهایت تابع wait از تابع sleep برای انتظار اتمام کار پردازه های فرزند استفاده میشود. در نتیجه اگر پردازه ای که حالت ZOMBIE داشته باشد پیدا نشود، پردازه والد با استفاده از sleep منتظر میماند.

```c
int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
    if(myproc()->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock); //DOC: piperead-sleep
  }
  for(i = 0; i < n; i++){  //DOC: piperead-copy
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);  //DOC: piperead-wakeup
  release(&p->lock);
  return i;
}
```

code from pipe.c

یکی از استفاده های sleep در pipe اتفاق می افتد چه در زمان خواندن از روی pipe و چه در زمان نوشتن اگر طرف مقابل هنوز اماده خواندن نباشد، و یا اطلاعاتی هنوز روی pipe برا خواندن موجود نیست.
در هر دو این حالت ها با استفاده از sleep پردازه منتظر میماند.

(15

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

Wakeup from proc.c

تابع wakeup وظیفه دارد که هر تابعی که با استفاده از sleep به حالت SLEEPING رفته بوده را بیدار کند.

(16

تابع sleep پردازه را از هر وضعیتی که هست به وضعیت SLEEPING میبرد.
در مرحله بعد تابع wakeup پردازه را از حالت SLEEPING → RUNNABLE میبرد.

```
//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(p->cal==EARLIEST_DEADLINE_FIRST) //additional
        number_of_runnable_processes_in_edf_queue++; //additional
      else if(p->cal==MULTILEVEL_FEEDBACK_QUEUE_FIRST_LEVEL) //additional
        number_of_runnable_multilevel_feedback_queue[0]++; //additional
      else if(p->cal==MULTILEVEL_FEEDBACK_QUEUE_SECOND_LEVEL) //additional
      {
        number_of_runnable_multilevel_feedback_queue[1]++; //additional
        p->entering_time_to_the_fcfs_queue=ticks; //additional
      }
    }
}
```

Wakeup1 from proc.c

```c
// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

kill from proc.c

همانطور که مشاهده میشود 2 تابع دیگر نیز درون proc.c وجود دارند که وضعیت پردازه را از SLEEPING به RUNNABLE میبرند.

(18

```
  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

Code from exit() in proc.c

قطعه کد بالا مربوط به تابع exit() میباشد، همانطور که دیده میشود زمانی که تابع exit برای یک پردازه صدا زده میشود. در پایان به تمامی فرزندانش یک والد جدید (در اینجا initproc) تخصیص داده میشود، این والد تا زمانی که به وضعیت تمامی فرزندان رسیدگی کند باقی خواهد ماند. تا در آینده با استفاده از wait منابع تمامی این فرزندان را آزاد کند.
پس در نتیجه درون xv6 پردازه orphan بوجود نمی آید و همگی همواره در یک چرخه وراثتی باقی میمانند.

(19

تمامی تعاملات کاربر با سیستم عامل از طریق shell اتفاق می افتند.
حال فرض میکنیم که تمامی پردازنده ها مشغول اجرای پردازه های مربوط به کلاس اول باشند. در نتیجه برای کلاس دوم پردازنده ای باقی نخواهد نماند و به عبارتی پردازه های درون کلاس دوم همگی starvation میکشند.
از آنجایی که shell نیز درون کلاس دوم در سطح اول قرار دارد، به آن نیز پردازنده ای تخصیص داده نمیشود چرا که همگی پردازنده ها مشغول رسیدگی به کلاس اول هستند.
به همین سبب کاربر مشاهده میکند که ترمینال فریز شده، این درحالیست که همگی پردازنده ها درحال اجرای پردازه های کلاس اول هستند.
برای جلوگیری از این موضوع می توان:

- زمان اجرای real-time یا همان کلاس اول را محدود کرد
- Shell بصورت اختصاصی پردازنده یا core خودش را داشته باشد

- با استفاده از aging این موضوع بصورت کلی هندل شود

(20

خیر، زمانی که مقدار CPUS = 2 باشد، دو هسته CPU خواهیم داشت که هر یک بصورت مستقل برای خودش تابع scheduler را اجرا میکند.
حال هر هسته میتواند از ptable (که مشترک است) پردازه های RUNNABLE را اجرا کند.
در نتیجه دیگر ترتیب RR حفظ نمیشود، به این دلیل که هر CPU برای خودش میتواند در یک نقطه متفاوتی از صف باشد.

پس درنتیجه دیگر ترتیب صف حفظ نمیشود چرا که ptable مشترک هست اما نوبت دهی CPU ها در اجرای پردازه ها با یکدیگر هماهنگ نیستند.

(21

از آنجایی که در Hard Real-time هیچگونه تاخیری پذیرفته نیست پس باید:
- پردازه ها باید بدون تاخیر باشند و در deadline خود به پایان برسند
- اگر پردازه ای به deadline اش نمیرسد باید از صف حذف شود

(22

تابع sleep وضعیت پردازه را به SLEEPING تغییر میدهد، پردازه در این وضعیت تنها منتظر یک I/O یا event هست که آن را بیدار کند یا به وضعیت RUNNABLE برگرداند. در نتیجه هیچ فعالیتی انجام نمیدهد.
به همین جهت در صف در زمان iteration از آن صرف نظر میشود، چرا که RUNNABLE نیست و قابلیت اجرا ندارد. و صرفا منتظر یک event خارجی میماند.

(23

این انتخاب باید از میان توابعی باشد که میتوانند در وضعیت پردازه تغییر ایجاد کنند یا به عبارتی آن را به RUNNABLE ببرند و یا از آن خارج کنند. پس با توجه به این موضوع توابع:
- scheduler
- fork , exec
- exit
- wakeup
- age_processes

در بخش های گذشته تک تک این توابع بررسی شدند، مشاهده شده که هر یک از این توابع در تغییر وضعیت یک پردازه میتوانند موثر باشند به همین دلیل در هر یک از این توابع باید تغییر ایجاد شود تا صف پردازه ها بروزرسانی شود.