بسم الله الرحمن الرحیم

پروژه سوم درس سیستم عامل

دکتر کارگهی

محمد امین توانایی - ۸۱۰۱۰۱۳۹۶

سید علی تهامی - ۸۱۰۱۰۱۳۹۷

مهدی وجهی - ۸۱۰۱۰۱۵۵۸

فهرست

سوال ۱

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

// 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)
};
```

۱. process state = state

۲. process number = pid

۳. Memory limits = sz

۴. list of open file = ofile

۵. program counter = context-> eip

سوال ۲

Runnable -> ready

Running -> running

Sleeping -> waiting

Zombie -> terminate

Embryo -> new

سوال ۳

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

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

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

```
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;
  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);
}
```

اگر پردازه init باشد توسط userinit این کار صورت میگیرد در غیر اینصورت اینکار توسط تابع fork انجام خواهد گرفت.

در ابتدا پردازه به حالت EMBRYO رفته و سپس به حالت runnable میرود دقیقا مثل همان چیزی که در شکل کتاب گفته شده است.

سوال ۴

```
#define NPROC        64  // maximum number of processes
#define KSTACKSIZE 4096  // size of per-process kernel stack
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
#define ROOTDEV       1  // device number of file system root disk
#define MAXARG       32  // max exec arguments
#define MAXOPBLOCKS  10  // max # of blocks any FS op writes
#define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
#define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
#define FSSIZE       1000  // size of file system in blocks
```

تعداد پردازه های قابل خلق به اندازه NPROC است که در اینجا ۶۴ است.

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

```
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;
```

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

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

در این حالت در آرایه ای از proc ها که در ptable نگه داری میشود استیت اولیه همه پردازه ها UNSEND است و وقتی پردازه جدیدی ایجاد میشود در ایندکس بعد جایی که یکی از عناصر آرایه runnable است پردازه جدید نیز اضافه شده و به حالت runnabble میرود.حال چون تا ایندکس ۶۴ حلقه ادامه پیدا میکند اگر نتواند در آن آرایه جای خالی پیدا کند(به این معنا که همه ۶۴ پردازه در حالت runnable باشند تابع allocproc صفر بر میگرداند و اگر صفر برگرداند تابع fork نیز با برگرداندن عدد ۱- به کاربر اطلاع میدهد که نمیتواند پردازه جدید ایجاد کند.

سوال ۵

اگر قبل از حلقه ptable را lock نکنیم ممکن است بصورت concurrent توسط kernel مورد دسترسی قرار بگیرد و دیتای آن عوض شود یا اینکه ممکن است حین schedule پردازه جدیدی وارد شود و در صورتی که نباید در زمانبندی شرکت داشته باشد شرکت داده میشود. همچنین ممکن است با قسمت های دیگه که با این داده کار می کنند و می خواهند روی آن کار کنند همروندی رخ دهد و مشکلاتی در داده به وجود بیاورد.

سوال ۶

در iteration بعدی، چون ساختار داده ptable توسط برنامه ریز قفل شده است.

سوال ۷

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

سوال ۸

Context -> eip

```
# 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
```

```
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;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
```

محتوای ثبات های قبلی را در پشته ذخیره می کند و محتوای ثبات های جدید را از پشته به داخل ثبات ها می آورد.

سوال ۹

عملا دیگر وقفه در برنامه ریزی موثر نبود و اجرای برنامه ها به صورت FIFO انجام می شد و دیگر اشتراک زمانی در بین آنها نداشتیم.

سوال ۱۰

Code: Interrupts

Devices on the motherboard can generate interrupts, and xv6 must set up the hardware to handle these interrupts. Devices usually interrupt in order to tell the kernel that some hardware event has occured, such as I/O completion. Interrupts are usually optional in the sense that the kernel could instead periodically check (or "poll") the device hardware to check for new events. Interrupts are preferable to polling if the events are relatively rare, so that polling would waste CPU time. Interrupt handling shares some of the code already needed for system calls and exceptions.

Interrupts are similar to system calls, except devices generate them at any time. There is hardware on the motherboard to signal the CPU when a device needs attention (e.g., the user has typed a character on the keyboard). We must program the device to generate an interrupt, and arrange that a CPU receives the interrupt.

Let's look at the timer device and timer interrupts. We would like the timer hardware to generate an interrupt, say, 100 times per second so that the kernel can track the passage of time and so the kernel can time-slice among multiple running processes. The choice of 100 times per second allows for decent interactive performance while not swamping the processor with handling interrupts.

پس با این شرایطی که در کتاب مرجع گفته شده هر ۱۰ میلی ثانیه یکبار timer interrupt صادر میشود.

سوال ۱۱

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)      You, 2 hours ago
  yield();
```

تابع trap این کار را انجام میدهد.

سوال ۱۲

Q = 10 millisecond

سوال ۱۳:

تابع wait در نهایت از تابع sleep استفاده می‌کند تا اجرای برنامه را تا زمان اتمام کار فرآیند فرزند متوقف کند. فرآیند والد، تابع sleep را فراخوانی می‌کند تا منتظر دریافت سیگنالی از فرزند باشد که نشان‌دهنده پایان کار آن

است. این مکانیسم از انتظار مشغول (busy waiting) جلوگیری می‌کند، که در آن والد به طور مداوم وضعیت فرزند را بررسی می‌کند و باعث مصرف غیرضروری سیکل‌های CPU می‌شود.

سوال ۱۴:

فراتر از استفاده آن در تابع wait، تابع sleep در سناریوهایی کاربرد دارد که یک فرآیند نیاز به توقف تا زمان برآورده شدن یک شرط خاص دارد. یکی از این موارد، هماهنگ‌سازی خواننده‌ها و نویسنده‌ها در pipe ها است. هنگامی که بافر pipe پر می‌شود، فرآیند نویسنده تابع sleep را فراخوانی می‌کند تا منتظر بماند تا یک خواننده مقداری از داده‌ها را مصرف کند و فضایی در بافر آزاد شود.

سوال ۱۵:

در کرنل xv6، تابعی که مسئول آگاه‌سازی یک فرآیند درباره رویدادی است که منتظر آن بوده، تابع wakeup است. هنگامی که event مورد انتظار رخ می‌دهد، تابع wakeup مربوطه فراخوانی می‌شود، فرآیند waiting را به عنوان RUNNABLE علامت‌گذاری می‌کند و به آن اجازه می‌دهد تا اجرای خود را از سر بگیرد.

سوال ۱۶:

برنامه را از وضعیت waiting به وضعیت ready منتقل میکند.

سوال ۱۷:

بله، تابع دیگری که می‌تواند باعث انتقال از حالت انتظار به حالت آماده شود، تابع yield است. یک فرآیند ممکن است داوطلبانه با استفاده از تابع yield، سی پی یو را رها کند. به عنوان مثال، فرآیندی که محاسبات طولانی انجام می‌دهد، می‌تواند به طور دوره‌ای yield را فراخوانی کند تا به سایر فرآیندها اجازه اجرا دهد. این کار تخصیص عادلانه CPU را تضمین می‌کند و از انحصار پردازنده توسط یک فرآیند واحد جلوگیری می‌کند. تابع yield به طور موثر فرآیند را از حالت RUNNING به RUNNABLE منتقل می‌کند، که این امکان را فراهم می‌سازد تا بعداً دوباره زمانبندی شود.

سوال ۱۸:

در xv6، زمانی که یک فرآیند والد خاتمه می‌یابد و فرآیندهای فرزند خود را به عنوان یتیم باقی می‌گذارد، فرآیند init این فرآیندهای یتیم را به فرزندی می‌پذیرد. فرآیند init، که یک فرآیند ویژه سیستمی است، به صورت دوره‌ای فراخوانی `wait` را انجام می‌دهد تا منابع این فرآیندهای یتیم (فرآیندهای زامبی) را پاکسازی کند. این اطمینان حاصل می‌شود که هیچ فرآیند زامبی به طور نامحدود در سیستم باقی نمی‌ماند و از نشت منابع جلوگیری می‌شود.

پیاده سازی

تنظیم کوانتوم زمانی RR

بتدا برنامه ای می نویسم که یک فرزند ایجاد می کند و همروند با فرزند هردو نام خود را چاپ می کنند. واضح است که اگر کوانتوم زمانی بزرگ تر باشد تعداد چاپ ها در هر اسلایس بیشتر است حالا ابتدا برنامه را با همان حالت پیش فرض تست می کنیم بخشی از نتیجه به صورت زیر است:

```
85 for paren107 for child
108 for childt

86 for parent
87 for paren109 for child
1t
1088 for p for child
arent
89 for111 for  parent
child
90 for parent
91 for parent112 for chi
92 forld
113 for child
```

```c
#include "types.h"
int main(int argc,char* argv[]){
    int pid = fork();
    if(pid < 0){
        return 1;
    }
    if(pid == 0){
        for (int i=0;i<1000000;i++){
            printf(1,"%d for child\n",i);
        }
    }
    else
    {
        for (int i=0;i<1000000;i++){
            printf(1,"%d for parent\n",i);
        }
    }
    return 0;
}
```

حال در تابع trap که با اتمام کوانتوم تعویض محتوا صورت می گیرد شرط می گذاریم که کوانتوم مضرب عددی باشد. این موضوع باعث می شود که کوانتوم ها به همان اندازه طولانی تر شوند. مثلا برای ۲۰ نتیجه به صورت زیر است:

```
142 for parent
143 for parent
123 for child
124 for child
125 for child
126 for child
127 for child
128 for child
129 for child
130 for child
131 for child
132 for child
133 for child
134 for child
135 for child
136 for child
137 for child
138 for child
139 for child
140 for child
144 for parent
145 for parent
```

```
 37   trap(struct trapframe *tf)
103     // Force process to give up CPU on clock tick
104     // If interrupts were on while locks held, wo
105     if(myproc() && myproc()->state == RUNNING &&
106        tf->trapno == T_IRQ0+IRQ_TIMER)
107       if(ticks % 20 == 0)
108         yield();
```

حال طبق خواسته سوال کوانتوم را رو ۵ تنظیم می کنیم.

سوال ۱۹

```
ifndef CPUS
CPUS := 2
```

با انجام این تغییر برنامه روی ۲ هسته به صورت موازی اجرا می شود و این موضوع باعث می شود که عملا تفاوتی بین کوانتوم های مختلف نباشد زیرا پردازه پدر و فرزند موازی و در ۲ هسته اجرا می شوند. (حتی با تغییر کوانتوم به ۱ ثانیه هم تفاوتی مشاهده نمی شود.)

```
93 f
81 for child
82 foorr  chiplad
83 rentfor chil
d
984 4f foor r parechintl
d
985 for child
586 for child
8 f7o rf opr cahirlentd
88
 9f6 or cfhor ild
parent
899 7 for parenfort
 9c8h ilfod
r90  forp cahreinld
t91
 f9or9 chi for parent
ld
9120 0fo fro r cpahild
r93 fore nt
ch1i0l1d
 for parent
102 for 9p4a rfeonrt
103 fcohild
```

SJF

```
76   allocproc(void)
90   found:
91     p->state = EMBRYO;
92     p->pid = nextpid++;
93     p->burst_time = 2;
94     p->confidence = 50;
```

ابتدا احتمال و زمان پیش بینی شده را به ساختمان داده پردازه ها اضافه می کنیم و سپس در هنگام ساخت مقدار دهی اولیه می کنیم. همچنین وقفه های زمانی هم غیر فعال می کنیم چون دیگر در اینجا کاربرد ندارد. سپس در تابع برنامه ریزی ابتدا لازم است بر اساس زمان های اتمام پردازه ها را مرتب کنیم. برای این کار تابع مرتب سازی را می نویسیم.

```
328  void sort_processes(void)
329  {
330    struct proc *p;
331    struct proc *arg_min;
332    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
333    {
334      if (p->state == SLEEPING)
335        continue;
336      arg_min = p;
337      for (struct proc *i = p + 1; i < &ptable.proc[NPROC]; i++)
338      {
339        if (i->state == SLEEPING)
340          continue;
341        if (arg_min->burst_time > i->burst_time && arg_min->pid != i->pid)
342        {
343          arg_min = i;
344        }
345      }
346      if (arg_min != p)
347      {
348        struct proc temp = *p;
349        *p = *arg_min;
350        *arg_min = temp;
351      }
352    }
353  }
```

حال اگر احتمال نداشتیم تنها کافی بود پیمایش کنیم و اولین پردازه را اجرا کنیم. ولی چون احتمال داریم بعد از پیدا کردن پردازه مربوطه عدد تصادفی ای ایجاد کرده و با احتمال مقایسه می کنیم اگر کمتر بود که پردازه اجرا می شود ولی اگر نبود به سراغ پردازه بعدی می رویم.

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

Fifo

در این قسمت همانند قسمت قبل نیازی به استفاده از yield نداریم و باید در ابتدای خلق پردازه زمان ورود را مقدار دهی کرده و در هر بار رفتن در تابع schedule بهترین کاندید برای اجرا (یعنی پردازه ای که کمترین زمان ورود را داراست) را پیدا کنیم و تا زمانی که exit نکند آن را اجرا میکنیم.

Aging

در این قسمت لازم هست هر پردازه ای که در استیت runnable هست در وقفه تایمر به اندازه یک واحد مسن شود این کار در تابع update_age انجام میشود.
همچنین در موقع تغییر صف یک پردازه لازم است سن آن پردازه صفر شود

```
void
update_age(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->state != RUNNABLE || p->pid < 3){
      p->age = 0;
      continue;}
    p->age++;
  }
  release(&ptable.lock);
}      mvajhi, 22 hours ago • Added enter time
```

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

تغییر صف

در هر وقفه تایمر چک میکنیم هر پردازه ای که بیش از حد مسن بود را به صف بالاتر انتقال میدهیم.

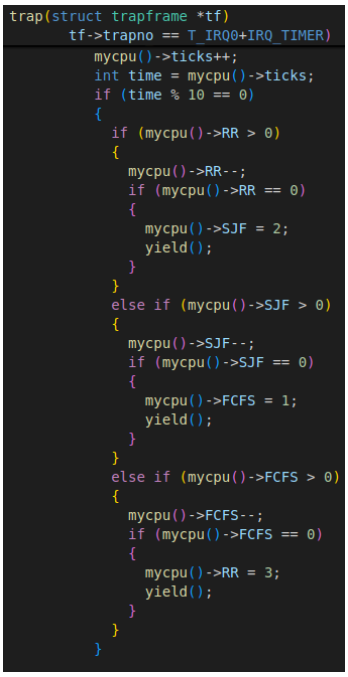
لازم به ذکر است دو تابع فوق اول وقفه تایمر صدا زده میشوند.

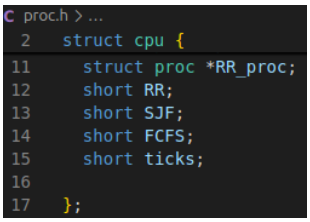
```
void
update_queue_number(void)
{
  acquire(&ptable.lock);
  for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    // TODO 80 -> 800
    if (p->queue_number == RR || p->age < 80 || p->pid < 3)
      continue;
    p->queue_number++;
    p->age = 0;
    p->enter_time = ticks;
   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

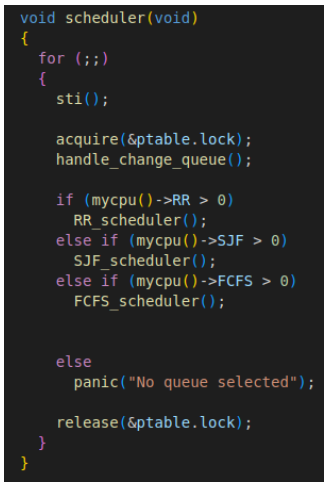
```
    if(myproc() && myproc()->state == RUNNING &&
       tf->trapno == T_IRQ0+IRQ_TIMER)
       {
          update_age();
          update_queue_number();
```

تجمیع

Time slicing

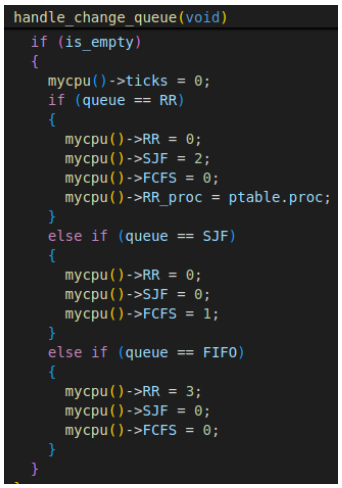






ابتدا به ساختمان داده cpu به ازای هر صف یک متغیر اضافه می کنیم که نشان دهنده زمان باقی مانده هست و هر کوانتوم زمانی خود را ۱۰۰ میلی ثانیه در نظر می گیریم و به ازای هر کوانتوم یکی واحد از زمان هر کدام را کم می کنیم. زمان ها هم طبق خواسته مسئله ۳،۲ و ۱ هست. در هر زمان که دیدیم زمان هر کدام هم تمام شده بعدی را مقدار دهی می کنیم و برای RR علاوه بر این کار پردازه در حال اجرا هم ذخیره می کنیم. همچنین در تابع برنامه ریزی هم نسبت به این که در کدام هستیم همان را برنامه ریزی می کنیم. نکته دیگر این که اگر در RR بودیم به ازای هر ۵ کوانتوم مجدد RR را صدا می کنیم.

هندل کردن اتمام زودتر از موعد



بعضی مواقع کار صف زودتر از موعد تمام می شود ولی نباید این زمان به صف بعدی داده شود برای هندل کردن این موضوع ما برای هر clock زمان سنج جدا گذاشتیم ولی در حالت عادی مانند ساعت اصلی افزایش می یابد ولی وقتی که وسط کار صف تمام می شود زمان آن صفر می شود که باقی مانده زمان به صفی نرسد یا از صفی زمانی کم نشود.

سوال ۲۰

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

سوال ۲۱

ما برای جابه جایی بین صف ها از روش اسلایس زمان استفاده کردیم و این موضوع مشکل قحطی در روش هر یک از سطح ها را برطرف نمی کند. به همین علت ما از روش aging استفاده می کنیم و پردازه ها را به سطوح بالاتر می آوریم و در بالاترین سطح از RR استفاده می کنیم که مشکل قحطی را ندارد.

سوال ۲۲

زیرا پردازه در انتظار رویدادی هست و به خواست خود غیر فعال است و اصلا آمادگی اجرا را ندارد پس با بالا رفتن در سطح ها تفاوتی برای آن رخ نمی دهد.