به نام خدا

گزارش آزمایشگاه آزمایش سوم درس سیستم عامل

آیدین کاظمی: 810101561 علی زیلوچی: 810101560 بابک حسینی محتشم: 810101408

شرح پروژه:

1. در فایل proc.h می‌توان ساختار PCB را مشاهده کرد:

```
35  enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
36
37  // Per-process state
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52    int sc[26]; //Babak          // Array storing the number of times each system call is invoked by this process
53  };
```

شکل 3-3 منبع درس:

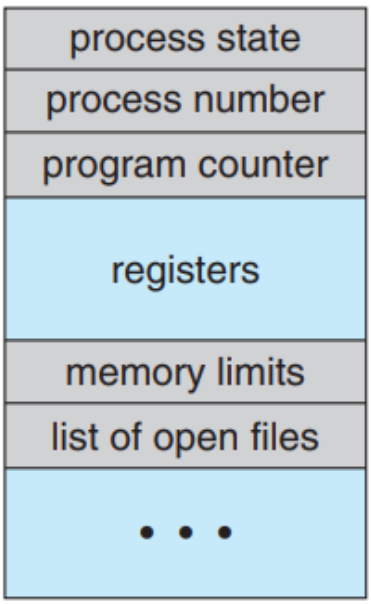


Figure 3.3 Process control block (PCB).

شباهت‌هایی بین این دو ساختار وجود دارد.

ساختار PCB در xv6	شکل 3-3 منبع درس
وضعیت پردازه در متغیر state ذخیره شده است.	process state
شماره پردازه در متغیر pid به صورت عدد صحیح ذخیره شده است.	process number
رجیسترها با مقادیرشان در context ذخیره شده‌اند.	registers
اندازه حافظه داده شده به پردازه برحسب بایت در sz ذخیره شده است.	memory limits
لیستی از فایل‌هایی که پردازه باز کرده در ofile ذخیره شده است.	list of open files

.2

توصیف در xv6	وضعیت در xv6	وضعیت معادل در شکل 1
وضعیت خانه‌هایی از جدول پردازه‌ها که پردازه‌ای در آن‌ها قرار ندارد.	UNUSED	new
پردازه تازه ساخته شده که هنوز آماده اجرا نیست و مقادیر PCB والد در PCB آن کپی نشده است.	EMBRYO	new
پردازه‌ای که آماده اجرا است و متنظر انتخاب شدن توسط زمان‌بند است.	RUNNABLE	ready
پردازه‌ای که در حال اجرا شدن است.	RUNNING	running
پردازه منتظر رویدادی خاص است.	SLEEPING	waiting
پردازه‌ای که اجرایش تمام شده ولی هنوز در جدول پردازه‌ها وجود دارد چون والد مقدار برگشتی آن را دریافت نکرده است.	ZOMBIE	terminated

حال برای این پروژه تغییراتی در داده ساختار های cpu و proc به وجود آمده که به شرح زیر است:

```
  struct proc *proc;          // The process running on this cpu or null
+ int _consecutive_runs_queue; // The number of times a process from the last queue has been
+ int _current_queue;         // The current queue the cpu is choosing processes from.
};
```

در ساختار cpu متغیر consecutive_runs_queue نشان دهنده تعداد دفعاتی است که یک متوالی از process های یکی از صفها استفاده کرده ایم، این متغیر برای ایجاد time scaling میان صف‌ها اضافه شده است و همچنین current_queue_ نیز برای نگه داری صف فعلی استفاده شده است.

```
  char name[16];               // Process name (debugging)
+ int sc[sizeof(syscall_names)/sizeof(char*)]; //Babak        // Array
+ int queue;                  // The scheduling queue
+ int wait_time;              // Total wait time
+ int confidence;             // Confidence in burst time
+ int burst_time;             // Burst time
+ int consecutive_runs;       // Last number of consecutive runs
+ int arrival;                // Time of arrival
};
```

در ساختار process متغیر queue که نشان دهنده صفی است که پردازه در آن قرار دارد، متغیر wait_time مدت زمان انتظار پردازه برای اجرا شدن را نشان می‌دهد (توجه شود که این متغیر بعد از تغییر صف، 0 خواهد شد)، متغیرهای confidence و burst_time برای الگوریتم SJF استفاده می‌شوند، متغیر consecutive_run نشان دهنده تعداد دفعات متوالی است که پردازه اجرا شده است و نهایتا arrival لحظه ورود پردازه را به صفوف نشان خواهد داد، اضافه شده‌اند.

تغییر وضعیت در xv6:

Admitted:

3. تابع allocproc در جدول پردازه‌ها به دنبال پردازه‌ای می‌گردد که وضعیت آن UNUSED باشد که یعنی آن پردازه آماده تولید شدن است. سپس وضعیت پردازه را به EMBRYO تغییر می‌دهد، شمارنده پردازه‌ها را یکی افزایش می‌دهد و در نهایت برخی مقداردهی‌های اولیه و ایجاد استک برای پردازه صورت می‌گیرد. تابع allocproc در دو جا صدا زده می‌شود. برای ایجاد اولین پردازه در تابع userinit که پس از ایجاد پردازه، آن را مقداردهی اولیه می‌کند و وضعیتش را به RUNNABLE تغییر می‌دهد. برای ایجاد بقیه پردازه‌ها در تابع fork پس از ایجاد پردازه، مقادیر پردازه پدر در PCB پردازه جدید کپی می‌شود و سپس وضعیت آن را به RUNNABLE تغییر می‌دهد.

در این بخش در مورد تغییرات تابع allocproc میتوان گفت:

```
  // which returns to trapret.
  sp -= 4;
  *(uint *)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context *)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  // Clear the system call history of the process.
  for (int i = 0; i < sizeof(p->sc) / sizeof(p->sc[0]); i+
    p->sc[i] = 0;
+ p->queue=0;
+ p->wait_time=0;
+ p->confidence=50;
+ p->burst_time=2;
+ p->consecutive_runs=0;
+ p->arrival=ticks;
  return p;
}
```

که در اینجا خطوط سبز رنگ برای مقدار دهی های اولیه پردازه تازه ساخته شده به کار میروند (صف ورودی همان صف اول باشد، زمان انتظار و تعداد اجرا صفر، مقادیر مربوط به sjf همان مقادیر پیش فرض و زمان ورود ثبت شود). حال در صورتی که پردازه تازه ساخته شده از فرزندان shell یا initproc نباشد، باید به صف با پایین ترین اولویت برود، که این در تابع fork و exec انجام شده است:

```
  acquire(&ptable.lock);

+ if(curproc->pid>2 && pid>2)
+   np->queue=2;
  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

:Schedular Dispatch

4. سقف تعداد پردازه‌های موجود در xv6 در متغیر NPROC در فایل param.h ذخیره شده است و مقدار پیش‌فرض آن 64 است. در تابع allocproc تمام خانه‌های جدول‌پردازه پیمایش می‌شود تا یک خانه خالی پیدا شود. اگر تعداد پردازه‌های فعلی برابر NPROC شود، allocproc خانه خالی‌ای پیدا نمی‌کند و صفر برمی‌گرداند و در تابع fork هم با دریافت صفر، منفی یک به نشانه خطا برمی‌گرداند.

5. چون در تابع scheduler تمام خانه‌های جدول‌پردازه را پیمایش می‌کند و پردازه‌ای را انتخاب می‌کند، برای اینکه در این حین تغییری در وضعیت پردازه‌ها رخ ندهد، جدول پردازه‌ها را قفل می‌کند. برای مثال ممکن است دو پردازنده یک پردازه را برای اجرا انتخاب کنند. در سیستم‌های تک پردازه‌ای نیز ممکن است حین اجرای تابع scheduler ممکن است وقفه‌ای رخ دهد و با اجرای ISR، تغییری در وضعیت پردازه‌ها صورت بگیرد پس در این حالت هم ممکن است مشکل پیش بیاید.

6. اگر از پردازه ذکر شده گذشته باشیم، بدیهی است که حتی اگر تغییر وضعیت هم نداده باشد در iteration بعدی دوباره آن را بررسی می‌کنیم. اگر هنوز به پردازه مورد نظر نرسیده باشیم، پس از رسیدن به آن در همین iteration اجرایش می‌کنیم. اگر هم به ایندکس پردازه مورد نظر رسیده بودیم، دو حالت وجود دارد. یا پس از بررسی شرط وضعیت پردازه، وضعیتش تغییر می‌کند که در آن صورت در iteration بعدی آن را اجرا می‌کنیم. همچنین ممکن است قبل از بررسی شرط وضعیت تغییر کند که در آن صورت در همان iteration آن را اجرا می‌کنیم. البته در عمل تغییر وضعیت حین پیمایش پردازه‌ها به دلیل قفل جدول‌پردازه‌ها صورت نمی‌گیرد.

Context Switch:

7. می‌توان با بررسی ساختارداده context در فایل proc.h لیست رجیسترهای این موجود در آن را پیدا کرد.

```
17  // Saved registers for kernel context switches.
18  // Don't need to save all the segment registers (%cs, etc),
19  // because they are constant across kernel contexts.
20  // Don't need to save %eax, %ecx, %edx, because the
21  // x86 convention is that the caller has saved them.
22  // Contexts are stored at the bottom of the stack they
23  // describe; the stack pointer is the address of the context.
24  // The layout of the context matches the layout of the stack in swtch.S
25  // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
26  // but it is on the stack and allocproc() manipulates it.
27  struct context {
28    uint edi;
29    uint esi;
30    uint ebx;
31    uint ebp;
32    uint eip;
33  };
```

8. رجیستر eip نشان دهنده program counter است. با هر بار صدا زدن تابع swtch در scheduler یا در yield، مانند صدا زدن هر تابع دیگر، program counter در استک push می‌شود. وقتی context switch انجام می‌شود، پوینتر استک تغییر می‌کند و این گونه esp به جایی از استک اشاره می‌کند که در آن program counter مورد نظر از طریق call ذخیره شده است.

Interrupt:

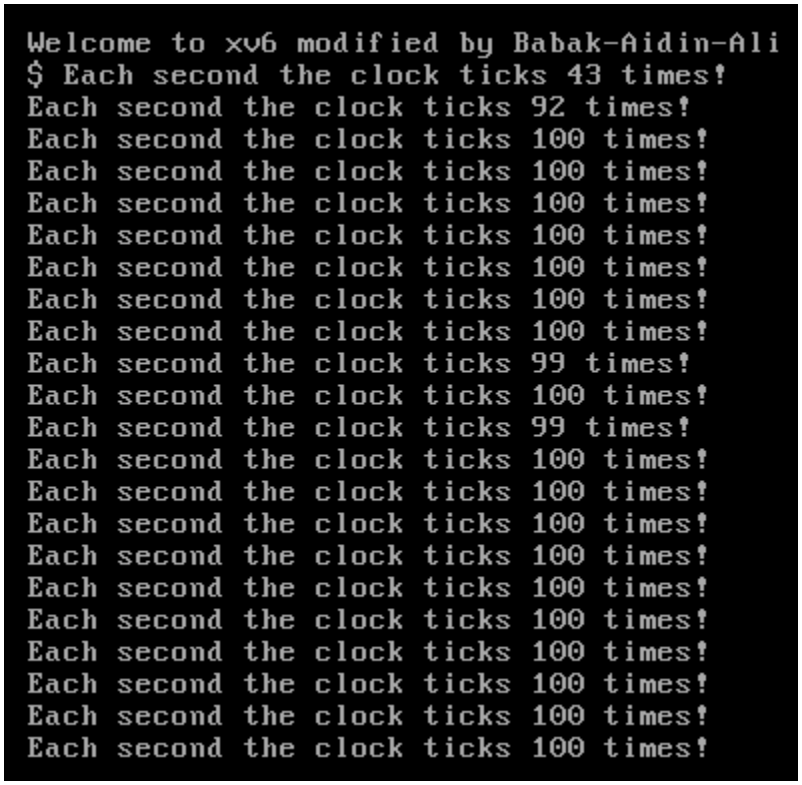
9. اگر interruptها پس از هر پیمایش، فعال نشوند ممکن است قبل از رسیدن به scheduler، interruptها غیرفعال شده باشند که در این صورت، قابلیت preemption که به timer interrupt وابسته است از بین می‌رود. مشکل بدتر این است که اگر پردازه‌ها روی رویدادی که از طریق interrupt اطلاع داده می‌شود مثل IO منتظر باشند، هیچ گاه این رویداد را دریافت نخواهند کرد و در صورتی که CPU در وضعیت idle باشد و پردازه قابل اجرایی نباشد، در همین وضعیت باقی می‌ماند چون پردازه‌ها منتظر باقی می‌مانند.
10. برای اندازه گیری اندازه هر tick، از تابع report_time_ را می‌نویسیم و پس از هر افزایش tick آن را صدا می‌زنیم. در این تابع دو متغیر static وجود دارد که یکی (last_ticks) برای ذخیره مقدار قبلی ticks و دیگری (last_time) برای ذخیره زمان قبلی است. با کمک تابع cmostime در فایل lapic.c زمان واقعی را می‌توان به دست آورد. پس با به دست آوردن زمان فعلی و حساب کردن اختلاف آن با last_time، می‌توانیم اختلاف زمانی دو tick را به دست آورده ولی چون زمان دریافت شده به واحد ثانیه است، تعداد tickها را در یک ثانیه به دست می‌آوریم.

```
void _report_time(){
  static uint last_ticks;
  static struct rtcdate last_time;
  struct rtcdate cur_time;
  cmostime(&cur_time);
  if(cur_time.second-last_time.second==1){
    cprintf("Each second the clock ticks %d times!\n",ticks-last_ticks);
    last_ticks=ticks;
  }
  last_time=cur_time;
}

//PAGEBREAK: 41
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++;
      _report_time();
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

می‌توان مشاهده کرد که به طور میانگین در هر ثانیه 100 باز timer interrupt صادر می‌شود پس با دقت خوبی فاصله زمانی هر دو tick حدود 10 میلی ثانیه است.



11. به طور متناوب در xv6 وقفه زمانی توسط سخت افزار داده می‌شود که این وقفه در تابع trap دریافت می‌شود. در این تابع با دریافت وقفه زمانی تابع yield صدا زده می‌شود. وضعیت پردازه از RUNNING به RUNNABLE تغییر می‌کند و سپس زمان‌بند اجرا می‌شود.

12. با توجه به نحوه کارکرد تابع scheduler، می‌توان نتیجه گرفت که با هر وقفه زمانی پردازه جدیدی در صورت امکان اجرا می‌شود پس time quantum برابر یک وقفه زمانی یا همان طول یک tick است که می‌شود 10 میلی ثانیه.

در این بخش لازم است اشاره کنیم که تابع yield را ابتدا با کمک تابع کمکی should yield به شکل زیر تغییر میدهیم:

```
int _should_yield(){
  struct proc *p = myproc();
  int queue_time_slice=time_slice*queue_weights[p->queue];
  if(++mycpu()->_consecutive_runs_queue>=queue_time_slice)
  {
    mycpu()->_consecutive_runs_queue=0;
    return 1;
  }
  switch (p->queue)
  {
  case 0:
    return (p->consecutive_runs>=5);
  case 1:
  case 2:
    return 0;

  default:
    return 1;
  }
}
```

```
// Give up the CPU for one scheduling round.
void yield(void)
{
  acquire(&ptable.lock); //DOC: yieldlock
  //cprintf("Pid: %d Consecutive runs: %d CPU: %d
  myproc()->consecutive_runs++;
  if(_should_yield()){
    myproc()->consecutive_runs = 0;
    myproc()->state = RUNNABLE;
    sched();
  }
  release(&ptable.lock);
}
```

که تابع مذکور در صورتی که برش زمانی به اتمام رسیده باشد یا در صف مربوط به RR باشیم (جلوتر توضیح داده خواهد شد) و کوانتوم زمانی به پایان رسیده باشد، یک بر میگرداند و موجب میشود عملیات عادی yield انجام شده و تعداد ران های متوالی هم ریست شود.

Wait:

13. در تابع wait ابتدا تمام پردازه‌های موجود در جدول‌پردازه‌ها بررسی می‌شوند تا پردازه‌های فرزند پیدا شوند. اگر پردازه فرزندی پیدا نشد این تابع منفی یک برمی‌گرداند. اگر پردازه فرزندی پیدا شد، بررسی می‌کند اگر پردازه فرزند در وضعیت ZOMBIE بود، آن را از جدول آزاد می‌کند و شماره و مقدار بازگشتی آن را به پردازه والد می‌دهد. اگر هیچ یک از پردازه‌های فرزند در وضعیت ZOMBIE نبودند، باید پردازه والد منتظر اتمام کار یکی از فرزندانش شود. برای این کار تابع sleep صدا زده می‌شود:

```
sleep(curproc, &ptable.lock);
```

14. برای مثال می‌توان از آن برای انتظار برای رویدادی خاص مثل وارد کردن input توسط کاربر استفاده کرد. برای مثال به همین منظور از آن در تابع consoleread استفاده می‌شود:

```
818  consoleread(struct inode *ip, char *dst, int n)
819  {
820    uint target;
821    int c;
822
823    iunlock(ip);
824    target = n;
825    acquire(&cons.lock);
826    while(n > 0){
827      while(input.r == input.w){
828        if(myproc()->killed){
829          release(&cons.lock);
830          ilock(ip);
831          return -1;
832        }
833        sleep(&input.r, &cons.lock);
834      }
835      c = input.buf[input.r++ % INPUT_BUF];
836      if(c == C('D')){  // EOF
837        if(n < target){
838          // Save ^D for next time, to make sure
839          // caller gets a 0-byte result.
840          input.r--;
841        }
842        break;
843      }
844      *dst++ = c;
845      --n;
846      if(c == '\n')
847      {
848        _history[_MOD(_last_history++,_N_HISTORY)]=input;
849        _current_history=_last_history;
850        struct _buffer new_input={"",0,0,0};
851        input=new_input;
852        break;
853      }
854    }
855    release(&cons.lock);
856    ilock(ip);
857
858    return target - n;
859  }
```

15. می‌توان از تابع wakeup برای اطلاع از رویدادی خاص به پردازه‌ها اطلاع داد. البته کار اصلی را تابع wakeup1 انجام می‌دهد و تابع wakeup تنها وظیفه قفل کردن جدول‌پردازه‌ها قبل از صدا زدن wakeup1 را دارد. در wakeup1 جدول‌پردازه‌ها پیمایش می‌شود و به ازای هر پردازه، وضعیت آن

را از SLEEPING به RUNNABLE تنها در صورتی تغییر می‌دهیم، که بدانیم پردازه منتظر همین رویداد بوده است و برای بررسی این موضوع، کافی است متغیر chan پردازه که در تابع sleep تعیین شد، با chan داده شده به wakeup مقایسه کنیم. هر پردازه‌ای می‌تواند با استفاده از wakeup پردازه دیگر را بیدار کند.

16. تابع wakeup منجر به گذار از SLEEPING به RUNNABLE در xv6 یا همان از waiting به ready در شکل کتاب می‌شود.

17. در تابع kill اگر پردازه‌ای که قصد خاتمه آن را داریم، در وضعیت SLEEPING باشد، وضعیت آن را به RUNNABLE تغییر می‌دهیم.

Exit:

18. در قسمتی از تابع exit والد تمام فرزندان را برابر initproc که پردازه ابتدایی است، قرار می‌دهیم و سپس wakeup1(initproc) را صدا می‌زنیم که باعث می‌شود پردازه initproc این پردازه‌های zombie را از جدول‌پردازه‌ها پاک کند.

زمان‌بندی بازخوردی چندسطحی:

سطح اول: زمان‌بندی نوبت گردشی با کوانتوم زمانی:

الگوریتم RR مد نظر به شکل زیر در فایل proc.c ایجاد شده است:

```
int _RR_scheduler(){
  static int index=0;
  for (int i=0; i<NPROC; i++)
  {
    index=(index+1)%NPROC;
    if (ptable.proc[index].state != RUNNABLE || ptable.proc[index].queue!=0)
      continue;
    return index;
  }
  return -1;
}
```

را دارد برای اجرا انتخاب می‌کند، توجه شود که global بودن متغیر index، موجب می‌شود حتی در صورت تغییر نوبت پردازه بین صفوف و اجرا شدن توابع دیگر ترتیب اجرای این الگوریتم برهم نخورد.

19. ابتدا الگوریتم سطح اول را برای برنامه تست با یک هسته اجرا می‌کنیم و خروجی زیر را مشاهده می‌کنیم. هر پردازه به اندازه پنج tick اجرا می‌شود و سپس preempt می‌شود.

```
Pid: 8 Consecutive runs: 2
Pid: 8 Consecutive runs: 3
Pid: 8 Consecutive runs: 4
Pid: 8 Consecutive runs: 5
Pid: 9 Consecutive runs: 1
Pid: 9 Consecutive runs: 2
Pid: 9 Consecutive runs: 3
Pid: 9 Consecutive runs: 4
Pid: 9 Consecutive runs: 5
Pid: 10 Consecutive runs: 1
Pid: 10 Consecutive runs: 2
Pid: 10 Consecutive runs: 3
Pid: 10 Consecutive runs: 4
Pid: 10 Consecutive runs: 5
Pid: 3 Consecutive runs: 1
Pid: 3 Consecutive runs: 2
Pid: 3 Consecutive runs: 3
Pid: 3 Consecutive runs: 4
Pid: 3 Consecutive runs: 5
Pid: 4 Consecutive runs: 1
Pid: 4 Consecutive runs: 2
Pid: 4 Consecutive runs: 3
Pid: 4 Consecutive runs: 4
Pid: 4 Consecutive runs: 5
Pid: 5 Consecutive runs: 1
Pid: 5 Consecutive runs: 2
Pid: 5 Consecutive runs: 3
Pid: 5 Consecutive runs: 4
Pid: 5 Consecutive runs: 5
Pid: 6 Consecutive runs: 1
Pid: 6 Consecutive runs: 2
Pid: 6 Consecutive runs: 3
Pid: 6 Consecutive runs: 4
Pid: 6 Consecutive runs: 5
Pid: 7 Consecutive runs: 1
Pid: 7 Consecutive runs: 2
Pid: 7 Consecutive runs: 3
Pid: 7 Consecutive runs: 4
Pid: 7 Consecutive runs: 5
Pid: 8 Consecutive runs: 1
Pid: 8 Consecutive runs: 2
```

حال تعداد CPUS را برابر دو قرار می‌دهیم و برنامه را دوباره اجرا می‌کنیم. می‌توان تاثیر موازی اجرا شدن CPUها را در خروجی زیر دید. دو پردازه متوالی نوبتی هر کدام پنج tick اجرا می‌شوند.

```
Pid: 10 Consecutive runs: 3 CPU: 1
Pid: 3 Consecutive runs: 1 CPU: 0
Pid: 10 Consecutive runs: 4 CPU: 1
Pid: 3 Consecutive runs: 2 CPU: 0
Pid: 10 Consecutive runs: 5 CPU: 1
Pid: 3 Consecutive runs: 3 CPU: 0
Pid: 4 Consecutive runs: 1 CPU: 1
Pid: 3 Consecutive runs: 4 CPU: 0
Pid: 4 Consecutive runs: 2 CPU: 1
Pid: 3 Consecutive runs: 5 CPU: 0
Pid: 4 Consecutive runs: 3 CPU: 1
Pid: 5 Consecutive runs: 1 CPU: 0
Pid: 4 Consecutive runs: 4 CPU: 1
Pid: 5 Consecutive runs: 2 CPU: 0
Pid: 4 Consecutive runs: 5 CPU: 1
Pid: 5 Consecutive runs: 3 CPU: 0
Pid: 6 Consecutive runs: 1 CPU: 1
Pid: 5 Consecutive runs: 4 CPU: 0
Pid: 6 Consecutive runs: 2 CPU: 1
Pid: 5 Consecutive runs: 5 CPU: 0
Pid: 6 Consecutive runs: 3 CPU: 1
Pid: 7 Consecutive runs: 1 CPU: 0
Pid: 6 Consecutive runs: 4 CPU: 1
Pid: 7 Consecutive runs: 2 CPU: 0
Pid: 6 Consecutive runs: 5 CPU: 1
Pid: 7 Consecutive runs: 3 CPU: 0
Pid: 8 Consecutive runs: 1 CPU: 1
Pid: 7 Consecutive runs: 4 CPU: 0
Pid: 8 Consecutive runs: 2 CPU: 1
Pid: 7 Consecutive runs: 5 CPU: 0
Pid: 8 Consecutive runs: 3 CPU: 1
Pid: 9 Consecutive runs: 1 CPU: 0
Pid: 8 Consecutive runs: 4 CPU: 1
Pid: 9 Consecutive runs: 2 CPU: 0
Pid: 8 Consecutive runs: 5 CPU: 1
Pid: 9 Consecutive runs: 3 CPU: 0
Pid: 10 Consecutive runs: 1 CPU: 1
Pid: 9 Consecutive runs: 4 CPU: 0
Pid: 10 Consecutive runs: 2 CPU: 1
Pid: 9 Consecutive runs: 5 CPU: 0
```

چون هر دو هسته به صورت موازی اجرا می‌شوند، هر یک، یکی از پردازه‌های موجود را به مدت پنج tick اجرا می‌کنند و به همین دلیل در خروجی یکی در میان یکی از cpuها پردازه‌اش را چاپ می‌کند.

حال دو روش دیگر را نیز در اینجا نشان میدهیم:

```
int _SJF_scheduler(){
  int p_idx[NPROC];
  int min_val=0,new_min=1e9;
  int idx = 0;
  while(idx<NPROC)
  {
    new_min=1e9;
    int flag=1;
    struct proc *p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->state != RUNNABLE || p->queue!=1)
        continue;
      if(min_val<p->burst_time){
        new_min=(new_min<p->burst_time) ? new_min : p->burst_time;
        flag=0;
      }
      else if(min_val==p->burst_time){
        p_idx[idx++]=p-ptable.proc;
        flag=0;
      }
    }
    min_val=new_min;
    if(flag)
      break;
  }
  static unsigned long int seed = 1;
  for (int i = 0; i < idx; i++)
  {
    int rand=((unsigned int)(seed / 65536) % 32768)%100;
    seed= (seed+ticks) * 1103515243 + 12345;
    if(rand<ptable.proc[p_idx[i]].confidence)
      return p_idx[i];
  }
  if(idx)
    return p_idx[idx-1];
  return -1;
}
```

این تابع ابتدا پردازه‌های صفش را که درخواست ورود به cpu دارند، بر حسب burst_time مرتب می‌کند؛ سپس روی لیست مرتب شده حرکت کرده و با ایجاد یک عدد تصادفی و مقایسه آن با conifidence پردازه متناظر، اولین پردازه‌ای که شریط مطلوب را دارد انتخاب می‌کند.

از طرفی برای fcfs:

```
int _FCFS_scheduler(){
  int min_val=1e9,min_idx=-1;
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state != RUNNABLE || p->queue!=2)
      continue;
    if(min_val>p->arrival)
    {
      min_val = p->arrival;
      min_idx = p-ptable.proc;
    }
  }
  return min_idx;
}
```

این تابع پردازه‌ای که زودتر وارد شده است، در صف متناظر قرار دارد و می‌خواهد اجرا شود را به عنوان پردازه منتخب اعلام می‌کند.

همچنین تابع scheduler به شکل زیر تغییر کرده است:

```
454     acquire(&ptable.lock);
455+    do
456+    {
457+      if(c->_consecutive_runs_queue==0)
458+        c->_current_queue=(c->_current_queue+1)%3;
459+      switch (c->_current_queue)
460+      {
461+      case 0:
462+        p_index=_RR_scheduler();
463+        break;
464+      case 1:
465+        p_index=_SJF_scheduler();
466+        break;
467+      case 2:
468+        p_index=_FCFS_scheduler();
469+        break;
470+
471+      default:
472+        p_index=_RR_scheduler();
473+        break;
474+      }
475+      if(p_index==-1)
476+      {
477+        c->_consecutive_runs_queue=0;
478+        if(c->_current_queue==_NQUEUE-1)
479+          break;
480         continue;
481+      }
482+      p=&ptable.proc[p_index];
483+      p->wait_time=0;
484       c->proc = p;
485       switchuvm(p);
486       p->state = RUNNING;
487       swtch(&(c->scheduler), p->context);
488       switchkvm();
489
490       // Process is done running for now.
491       // It should have changed its p->state before coming back.
492       c->proc = 0;
493+    }while (c->_consecutive_runs_queue || c->_current_queue!=_NQUEUE-1);
494     release(&ptable.lock);
```

تغییرات ایجاد شده برای اعمال زمان بندی بر روی هر سه صف با الگوریتم های متفاوت میباشد (تغییر صف و صدا زدن تابع مربوط به حرف صف انجام شده) که در صورتی که بررسی همه صف ها به اتمام برسد، زمانبند از حلقه خارج شده و کار آن به اتمام میرسد تا دوباره صدا شود.

برش‌دهی زمانی:

20. نیاز داریم مقداردهی اولیه CPUها در ابتدای کار صورت بگیرد. ما مقدار دهی اولیه را قبل از شروع به کار CPUها در تابع mpmain انجام دادیم چون در ابتدای کار تمام CPUها وارد این تابع می‌شوند پس همه به درستی مقداردهی اولیه می‌شوند.

```
static void
mpmain(void)
{
  mycpu()->_consecutive_runs_queue=0;
  mycpu()->_current_queue=2;
  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
}
```

21. برای پردازه‌های موجود در صف سوم به دلیل این‌که ابتدا پردازه‌ای که زودتر وارد شده باید کامل اجرا بشه بعد بقیه پردازه‌ها اجرا شوند، به همین دلیل اگر یکی از پردازه‌ها به شکلی block شود و زمان اجرای آن تا ابد طول بکشد این باعث می‌شود که بقیه پردازه‌های موجود در صف نیز نتوانند اجرا شوند. در مورد صف دوم هم چون امکان دارد مدام پردازه‌ای با burst time کم‌تر وارد شود و همواره عدد تصادفی ساخته شده کوچکتر از confidence آن باشد، برای پردازه‌ای starvation رخ دهد.

سازوکار افزایش سن:

22. زمانی که یک پردازه در وضعیت sleeping است، تمایلی برای گرفتن CPU ندارد و برای همین زمان انتظار آن را افزایش نمی‌دهیم. از طرف دیگر پردازه با وضعیت sleeping منتظر عوامل خارجی است که مشخص نیست که رخ دهند و به همین دلیل ممکن است برای این عوامل بسیار منتظر بماند و صف آن تغییر کند در صورتی که بقیه پردازه‌ها که در این مدت منتظر CPU بودند و اجرا شدند و معمولا CPU-bound هستند، تعداد دفعاتی اجرا شوند و تغییر صف ندهند در صورتی که مدت بیشتری منتظر اجرا روی CPU بوده‌اند.

در مورد تابع aging میتوان به نکات زیر اشاره کرد:

```
void _aging()
{
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->state==RUNNABLE && ++p->wait_time>=MAX_WAIT_TIME && p->queue)
    {
      cprintf("Process: %d has been moved from queue %d to queue %d due to aging.\n",p->pid,p->queue,p->queue-1);
      p->queue--;
      p->arrival=ticks;
      p->wait_time=0;
    }
  }
  release(&ptable.lock);
  return;
}
```

این تابع به ازای تمام پراسس ها، چک میکند که آیا زمان انتظار آنها در صف مربوطه بیش از زمان انتظار تعیین شده میباشد یا خیر، و در این صورت صف پردازه را یکی کمتر کرده (اولویت بالاتر برده)، مقدار زمان انتظار را صفرکرده و زمان رسیدن را آپدیت میکند. در صورتی که پردازه به بالاتر سطح اولویت برسد، دیگر اتفاقی نمی افتد.

سیستم کال های اضافه شده به صورت زیر است:

فراخوانی سیستمی set_sjf_info برای مقداردهی زمان اجرا و سطح اطمینان در هر پردازه است تا بتوان از این موارد برای الگوریتم SJF بهره برد:

```
int set_sjf_info(int pid,int burst,int confidence)
{
  struct proc *p;
  acquire(&ptable.lock);
  if(pid<=0 || pid>=NPROC)
  {
    cprintf("Invalid pid\n");
    return -1;
  }
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      p->burst_time=burst;
      p->confidence=confidence;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

فراخوانی سیستمی set_queue برای تعیین صف پردازه است، که در جابجایی بین صفوف زمانبندی کاربرد خواهد داشت:

```c
int set_queue(int pid,int queue)
{
  if(pid<=0 || pid>=NPROC)
  {
    cprintf("Invalid pid\n");
    return -1;
  }
  if(queue > 3 || queue < 0)
  {
    cprintf("Invalid queue\n");
    return -1;
  }
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      if(p->queue==queue)
      {
        cprintf("The process with pid %d is already in queue %d\n",pid,queue);
        return -1;
      }
      p->queue=queue;
      p->arrival=ticks;
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

فراخوانی سیستمی report_all_processes برای چاپ اطلاعات به صورت گفته شده در صورت پروژه استفاده می‌شود:

```c
int report_all_processes(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  cprintf("Name\tPid\tState\tQueue\tWait time\tConfidence\tBurst time\tConsecutive runs\tArrival\n");
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid==UNUSED)
      continue;
    cprintf("%s\t%d\t%s\t%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n",
    p->name,p->pid,states_names[p->state],p->queue,p->wait_time,p->confidence,p->burst_time,p->consecutive_runs,p->arrival);
  }
  release(&ptable.lock);
  return 0;
}
```

خروجی این تست به صورت زیر خواهد بود:

Name	Pid	State	Queue	Wait time	Confidence	Burst time	Consecutive runs	Arrival
init	1	sleep	0	0	50	2	1	0
sh	2	sleep	0	0	50	2	3	15
test	3	sleep	2	0	50	2	0	318
test	4	sleep	2	0	50	2	5	325
test	5	sleep	2	0	50	2	6	330
test	6	sleep	2	0	50	2	0	330
test	7	sleep	2	0	50	2	6	330
test	8	sleep	2	0	50	2	4	330
test	9	sleep	2	0	50	2	3	335
test	10	sleep	1	0	50	2	0	1145
	31	embryo	0	0	50	2	0	1987
test	32	run	2	0	50	2	2	2141
test	13	sleep	1	0	50	2	15	1369
test	14	sleep	1	0	50	2	1	1414
test	15	runble	1	8	50	2	0	1443
test	16	runble	1	70	50	2	0	1493
test	17	runble	1	225	50	2	0	1493
test	19	sleep	2	0	50	2	0	872
test	20	runble	1	131	50	2	0	1718
test	21	runble	0	17	50	2	0	2625
test	22	runble	0	24	50	2	0	2626
test	23	runble	0	21	50	2	0	2722
test	24	runble	1	1	50	2	0	1956
test	25	run	2	0	50	2	8	1386
test	26	runble	1	576	50	2	0	2189
test	27	runble	1	174	50	2	0	2591
test	28	runble	1	296	50	2	0	2469
test	29	runble	1	203	50	2	0	2562
test	30	runble	1	183	50	2	0	2582
test	33	runble	2	82	50	2	0	2647

لازم به ذکر است تمام این سیستم کال ها مانند پروژه های گذشته، در فایل های sysproc.c, user.h, usys.S و syscall.c آورده شده اند که در ادامه تصویر مربوط به فایل sysproc.c را مشاهده میکنید:

```
130+ // Set confidence and burst time for a process
131+ int
132+ sys_set_sjf_info(void)
133+ {
134+   int pid,burst,confidence;
135+   if(argint(0, &pid) < 0)
136+     return -1;
137+   if(argint(1, &burst) < 0)
138+     return -1;
139+   if(argint(2, &confidence) < 0)
140+     return -1;
141+   return set_sjf_info(pid,burst,confidence);
142+ }
143+
144+ // Set queue number of a process
145+ int
146+ sys_set_queue(void)
147+ {
148+   int pid,queue;
149+   if(argint(0, &pid) < 0)
150+     return -1;
151+   if(argint(1, &queue) < 0)
152+     return -1;
153+   return set_queue(pid,queue);
154+ }
155+
156+ // Print information about a process given its pid
157+ int
158+ sys_report_all_processes(void)
159+ {
160+   return report_all_processes();
161+ }
```

تست کلی برنامه:

برای تست کردن کارکرد کلی الگوریتم MLFQ، برنامه زیر را اجرا می‌کنیم و ورودی 0 را به برنامه میدهیم:

```c
void heavy_calculation(){
  for (int i = 0; i < 1e8; i++);
}
void ca3_test(int argc, char *argv[]){
  if (argc<2)
  {
    printf(2, "usage: test algorithm...\n");
    exit();
  }
  if (!strcmp(argv[1],"0"))
    report_all_processes();
  else if (!strcmp(argv[1],"rr")){
    for (int i = 0; i < 5; i++)
      fork();
    // report_all_processes();
    heavy_calculation();
    for (int i = 0; i < 5; i++)
      wait();
  }
  else if (!strcmp(argv[1],"sjf")){
    int pids[4],bursts[4]={6,3,4,7},confidences[4]={50,50,50,50};
    if((fork())==0){
      heavy_calculation();
      exit();
    }
    for (int i = 0; i < 4; i++)
    {
      if((pids[i]=fork())==0)
      {
        for (int i = 0; i < 1e4; i++);
        exit();
      }
      else
        set_sjf_info(pids[i],bursts[i],confidences[i]);
    }
  }
  else if (!strcmp(argv[1],"set_sjf_info"))
    set_sjf_info(atoi(argv[2]),atoi(argv[3]),atoi(argv[4]));
  else if (!strcmp(argv[1],"set_queue"))
    set_queue(atoi(argv[2]),atoi(argv[3]));
  else if (!strcmp(argv[1],"report_all"))
    report_all_processes();
  exit();
}
int
main(int argc, char *argv[]) {
  if(!strcmp(argv[argc-1],"2"))
    ca2_test(argc-1,argv);
  else if(!strcmp(argv[argc-1],"3"))
    ca3_test(argc-1,argv);
  else
    printf(2, "usage: test ... ca\n");
  exit();
}
```

با ورودی صفر در این برنامه تعداد زیادی پردازه ایجاد می‌کنیم. سپس تمام این پردازه‌ها وارد حلقه for طولانی می‌شوند و در نهایت روی بقیه پردازه‌ها wait می‌کنند. همچنین برای مشاهده پدیده aging، تعداد سیکل‌های مورد نیاز برای رخ دادن aging را از 800 به 300 کاهش می‌دهیم.

تصویر زیر در خروجی مشاهده شده است. می‌توان مشاهده کرد که در ابتدا CPU صفرم در حال اجرای صف 0 و CPU یک در حال اجرای صف اول است. همچنین non_preemptive بودن صف اول مشخص است. CPU صفرم پس از اجرای صف صفرم به مدت 300 میلی ثانیه صف بعدی را اجرا می‌کند. همچنین CPU اول پس از 200 می‌ثانیه صف آخر را اجرا می‌کند. در این حین پردازه شماره 12 از صف 1 به صف 0 به دلیل aging منتقل می‌شود. به همین ترتیب برنامه ادامه پیدا می‌کند و می‌توان دید صف آخر نیز non_preemptive است و پس از اجرا به مدت 100 میلی ثانیه CPU اول به صف صفرم بازمی‌گردد:

```
Pid: 21 Consecutive runs process: 2 CPU: 0 Queue: 0 Consecutive runs CPU: 27
Pid: 8 Consecutive runs process: 9 CPU: 1 Queue: 1 Consecutive runs CPU: 9
Pid: 8 Consecutive runs process: 10 CPU: 1 Queue: 1 Consecutive runs CPU: 10
Pid: 21 Consecutive runs process: 3 CPU: 0 Queue: 0 Consecutive runs CPU: 28
Pid: 8 Consecutive runs process: 11 CPU: 1 Queue: 1 Consecutive runs CPU: 11
Pid: 8 Consecutive runs process: 12 CPU: 1 Queue: 1 Consecutive runs CPU: 12
Pid: 8 Consecutive runs process: 13 CPU: 1 Queue: 1 Consecutive runs CPU: 13
Pid: 21 Consecutive runs process: 4 CPU: 0 Queue: 0 Consecutive runs CPU: 29
Pid: 8 Consecutive runs process: 14 CPU: 1 Queue: 1 Consecutive runs CPU: 14
Pid: 11 Consecutive runs process: 0 CPU: 0 Queue: 1 Consecutive runs CPU: 0
Pid: 8 Consecutive runs process: 15 CPU: 1 Queue: 1 Consecutive runs CPU: 15
Pid: 8 Consecutive runs process: 16 CPU: 1 Queue: 1 Consecutive runs CPU: 16
Pid: 8 Consecutive runs process: 17 CPU: 1 Queue: 1 Consecutive runs CPU: 17
Pid: 8 Consecutive runs process: 18 CPU: 1 Queue: 1 Consecutive runs CPU: 18
Pid: 11 Consecutive runs process: 1 CPU: 0 Queue: 1 Consecutive runs CPU: 1
Pid: 8 Consecutive runs process: 19 CPU: 1 Queue: 1 Consecutive runs CPU: 19
Pid: 6 Consecutive runs process: 0 CPU: 1 Queue: 2 Consecutive runs CPU: 0
Pid: 11 Consecutive runs process: 2 CPU: 0 Queue: 1 Consecutive runs CPU: 2
Pid: 6 Consecutive runs process: 1 CPU: 1 Queue: 2 Consecutive runs CPU: 1
Pid: 6 Consecutive runs process: 2 CPU: 1 Queue: 2 Consecutive runs CPU: 2
Process: 12 has been moved from queue 1 to queue 0 due to aging.
Pid: 6 Consecutive runs process: 3 CPU: 1 Queue: 2 Consecutive runs CPU: 3
Pid: 11 Consecutive runs process: 3 CPU: 0 Queue: 1 Consecutive runs CPU: 3
Pid: 6 Consecutive runs process: 4 CPU: 1 Queue: 2 Consecutive runs CPU: 4
Pid: 6 Consecutive runs process: 5 CPU: 1 Queue: 2 Consecutive runs CPU: 5
Pid: 11 Consecutive runs process: 4 CPU: 0 Queue: 1 Consecutive runs CPU: 4
Pid: 11 Consecutive runs process: 5 CPU: 0 Queue: 1 Consecutive runs CPU: 5
Pid: 6 Consecutive runs process: 6 CPU: 1 Queue: 2 Consecutive runs CPU: 6
Pid: 11 Consecutive runs process: 6 CPU: 0 Queue: 1 Consecutive runs CPU: 6
Pid: 6 Consecutive runs process: 7 CPU: 1 Queue: 2 Consecutive runs CPU: 7
Pid: 6 Consecutive runs process: 8 CPU: 1 Queue: 2 Consecutive runs CPU: 8
Pid: 11 Consecutive runs process: 7 CPU: 0 Queue: 1 Consecutive runs CPU: 7
Pid: 6 Consecutive runs process: 9 CPU: 1 Queue: 2 Consecutive runs CPU: 9
Pid: 11 Consecutive runs process: 8 CPU: 0 Queue: 1 Consecutive runs CPU: 8
Pid: 12 Consecutive runs process: 0 CPU: 1 Queue: 0 Consecutive runs CPU: 0
Pid: 11 Consecutive runs process: 9 CPU: 0 Queue: 1 Consecutive runs CPU: 9
Pid: 12 Consecutive runs process: 1 CPU: 1 Queue: 0 Consecutive runs CPU: 1
Pid: 12 Consecutive runs process: 2 CPU: 1 Queue: 0 Consecutive runs CPU: 2
Pid: 11 Consecutive runs process: 10 CPU: 0 Queue: 1 Consecutive runs CPU: 10
Pid: 12 Consecutive runs process: 3 CPU: 1 Queue: 0 Consecutive runs CPU: 3
```