به نام خدا

گزارش آزمایشگاه سیستم عامل

آزمایش سوم(بخش code)

سید نوید هاشمی 810101549

سید محمد جزایری 810101399

سبحان کوشکی جهرمی 810101496

سطح اول-زمانبند نوبت گردشی با کوانتوم زمانی

برای اینکه بتوانیم سطوح مختلف را هندل کنیم، یک متغیر در PCB اضافه کردیم که صف کنونی پردازه را نگه می دارد و آن level_queue نام دارد.

```
//This is for RR
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(mycpu()->RR)
  {
    if(check_no_one_in_queue(0))
    {
      mycpu()->RR = 0;
      mycpu()->SJF = 2;
      break;
    }
    if((p->state != RUNNABLE) || (p->level_queue != 0))
      continue;

    int min_last_exec = find_min_last_exec();
    if(p->last_exec == min_last_exec)
    {
      c->proc = p;
      // cprintf("TICKS: %d PID: %d exec...\n", ticks, p->pi
      switchuvm(p);
      p->state = RUNNING;

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

      c->proc = 0;
    }
  }
}
```

با توجه به کدی که قرار داده شده برای این بخش با توجه به اینکه پیش فرض خود xv6 همین الگوریتم بود کد خودش را با یکسری تغییرات استفاده کردیم در واقع با استفاده از تابع find_min_last_exec هر بار یک پردازه اجرا می شود یک متغیر last_exec دارد و چک میکنیم و آن پردازه ای که آخرین اجرایش دیر تر از بقیه باشد را می یابیم و اجرا میکنیم و در رابطه با کوانتوم زمانی که در RR وجود دارد، در فایل trap با توجه به اینکه هر tick 10 میلی ثانیه است یک متغیر را با مقدار اولیه 5 تعیین می کنیم و پس از 5 بار tick خوردن، متوجه می شویم که باید این پردازه را از حالت اجرا خارج کنیم که در تصاویر زیر مشخص است.

با هر بار tick خوردن سیستم num_of_interr را یکی کم میکنیم و در دو تصویر بعدی واضح است که اگر مقدارش 0 شد آن را دوباره به مقدار 5 برمیگردانیم و yield را صدا می زنیم تا پردازنده از پردازه فعلی گرفته شود و حالت RR را شبیه سازی کنیم.(در واقع مقدار NUM_OF_ITERR را 5 گذاشتیم.)

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
{
  //RR
  if(!check_no_one_in_queue(0))
  {
    if(time_slice <= 0)
    {
      time_slice = TIME_SLICE;
      if(mycpu()->RR > 0)
      {
        mycpu()->RR--;
        if((mycpu()->RR == 0) || (check_no_one_in_queue(0)))
        {
          // cprintf("RR time is over %d.\n", mycpu()->RR);
          mycpu()->SJF = 2;
          yield();
        }
      }
    }
    if(num_of_interr <= 0)
    {
      yield();
      num_of_interr = NUM_OF_ITERR;
    }
  }
```

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    handle_aging_and_consecutive();
    num_of_interr--;
    time_slice--;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
case T_IRQ0 + IRQ_IDE:
  ideintr();
```

```
allocproc(void)
  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;
  p->syscall_count = 0;
  p->arrival = ticks;
  p->last_exec = ticks;
  p->burst = DEFAULT_BURST;
  p->certainty = DEFAULT_CERTAINTY;
  p->queue_arrival = ticks;
  p->consecutive = 0;

  if((p->pid == 1) || (p->pid == 2))
  {
    p->level_queue = 0;
  }
  else if((p->parent->pid == 1) || (p->parent->pid == 2))
  {
    p->level_queue = 0;
  }
  else
  {
    p->level_queue = 0;
  }
```

برای تست درستی نیز همه پردازه ها را زمانی که ایجاد می شوند در صف اول قرار می دهیم(همانطور که اشاره کردم متغیر level_queue را برای این امر تعریف کردیم) تا همه RR اجرا شوند.

با اجرای برنامه سطح کاربر خودمان خروجی زیر را خواهیم داشت که روند اجرای الگوریتم RR را به وضوح نشان می دهد:

```
TICKS: 315 PID: 12 exec...
TICKS: 316 PID: 13 exec...
TICKS: 317 PID: 14 exec...
TICKS: 317 PID: 15 exec...
TICKS: 318 PID: 16 exec...
TICKS: 319 PID: 17 exec...
TICKS: 320 PID: 18 exec...
TICKS: 321 PID: 4 exec...
TICKS: 322 PID: 5 exec...
TICKS: 322 PID: 6 exec...
TICKS: 323 PID: 7 exec...
TICKS: 323 PID: 8 exec...
TICKS: 323 PID: 9 exec...
TICKS: 325 PID: 10 exec...
TICKS: 328 PID: 11 exec...
TICKS: 331 PID: 12 exec...
TICKS: 332 PID: 13 exec...
TICKS: 332 PID: 15 exec...
TICKS: 334 PID: 16 exec...
TICKS: 334 PID: 17 exec...
TICKS: 337 PID: 18 exec...
TICKS: 339 PID: 12 exec...
TICKS: 339 PID: 2 exec...
```

سطح سوم-اولین ورود اولین رسیدگی

```
//FCFS
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(mycpu()->FCFS)
  {
    // cprintf("SJF: %d\n", mycpu()->SJF);
    if(!check_no_one_in_queue(2))
    {
      if((p->state != RUNNABLE) || (p->level_queue != 2))
        continue;
      int first_come = find_first_come();
      if(p->arrival == first_come)
      {
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

        swtch(&(c->scheduler), p->context);
        c->proc = 0;
      }
    }
    else
    {
      mycpu()->FCFS = 0;
      mycpu()->RR = 3;
    }
  }
}
```

متغیری تحت عنوان arrival در PCB هر پردازه قرار دادیم که زمان ورود به صف را برای این الگوریتم نشان می دهد که در ابتدا وقتی fork می شود زمان ticks را برابر آن قرار می دهیم.سپس با توجه به تابع find_first_come پردازه ای که زودتر از بقیه وارد صف شده است را می یابیم و پردازنده را در اختیار آن قرار میدهیم.

در ادامه در فایل trap.c باید کاری کنیم که دیگر اینتراپت تایمر باعث تغییر پردازه نشود و تا وقتی پایان نیافته cpu از آن گرفته نشود که کد آن در عکس زیر قرار دارد. در واقع همانطور که دیده می شود دیگر با interrup timer تابع yield صدا زده نمی شود(در دو موردی که این تابع صدا زده شده است مربوط به بخش های آتی است که هر سه سطح با هم ترکیب می شوند در واقع تابع yield به نیت تغییر پردازه در همان صف صدا زده نشده است.)

```
//SJF and FCFS
else if(mycpu()->RR == 0)
{
  time_slice = TIME_SLICE;
  if(mycpu()->SJF > 0)
  {
      mycpu()->SJF--;
      if((mycpu()->SJF == 0) || (check_no_one_in_queue(1)))
      {
        // cprintf("SJF time is over %d.\n", mycpu()->SJF);
        mycpu()->FCFS = 1;
        yield();
      }
  }
  if(mycpu()->FCFS > 0)
  {
    mycpu()->FCFS--;
    if((mycpu()->FCFS == 0) || (check_no_one_in_queue(2)))
    {
      // cprintf("FCFS time is over %d.\n", mycpu()->FCFS);
      mycpu()->RR = 3;
      yield();
    }
  }
```

برای بررسی درستی همانطور که در مورد سطح اول گفته شد، پردازه ها را بجز shell و init همه را به صف سوم assign می کنیم و همانطور که در تصویر زیر که خروجی است مشاهده می شود یک پردازه در این صف وقتی پردازنده را در اختیار می گیرد تا وقتی تمام شود آن را در اختیار خواهد داشت:

```
if((p->pid == 1) || (p->pid == 2))
{
  p->level_queue = 0;
}
else if((p->parent->pid == 1) || (p->parent->pid == 2))
{
  p->level_queue = 0;
}
else
{
  p->level_queue = 2;
}
```

```
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 15 BY ARRIVAL 275
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 16 BY ARRIVAL 275
FIRST COME IS 16 BY ARRIVAL 275
FIRST COME IS 16 BY ARRIVAL 275
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 17 BY ARRIVAL 275
FIRST COME IS 17 BY ARRIVAL 275
FIRST COME IS 17 BY ARRIVAL 275
FIRST COME IS 17 BY ARRIVAL 275
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 3 BY ARRIVAL 262
FIRST COME IS 18 BY ARRIVAL 276
FIRST COME IS 3 BY ARRIVAL 262
```

برش دهی زمانی

در این بخش باید هر سه صف را بطور اشتراکی و همانگونه که در صورت پروژه گفته شده است ادغام کنیم با توجه به کدهایی که تصاویر آن پایین قابل مشاهده است ما در ابتدا یه متغیر در struct cpu با عناوین RR, SJF, FCFS قرار دادیم و در ابتدا مقادیر آنها را بجز برای RR=2 برابر 0 گذاشتیم.کاری که در این بخش میکنیم این است از آنجا که هر اسلایس زمانی 100 میلی ثانیه است ما باید 10 بار ticks را بشماریم که برای این کار متغیر time_slice را در ابتدا برابر 10 میگذاریم و هر بار یکی از آن کم میکنیم و هر موقع مقدارش 0 شد با توجه به اینکه در کدام صف هستیم مقادیر RR, SJF, FCFS را بروزرسانی کرده و دوباره آن را برابر 10 قرار می دهیم.

```
//RR
if(!check_no_one_in_queue(0))
{
  if(time_slice <= 0)
  {
    time_slice = TIME_SLICE;
    if(mycpu()->RR > 0)
    {
      mycpu()->RR--;
      if((mycpu()->RR == 0) || (check_no_one_in_queue(0)))
      {
        // cprintf("RR time is over %d.\n", mycpu()->RR);
        mycpu()->SJF = 2;
        yield();
      }
    }
  }
  if(num_of_interr <= 0)
  {
    yield();
    num_of_interr = NUM_OF_ITERR;
  }
}
//SJF and FCFS
else if(mycpu()->RR == 0)
{
  time_slice = TIME_SLICE;
  if(mycpu()->SJF > 0)
  {
      mycpu()->SJF--;
      if((mycpu()->SJF == 0) || (check_no_one_in_queue(1)))
      {
        // cprintf("SJF time is over %d.\n", mycpu()->SJF);
        mycpu()->FCFS = 1;
        yield();
      }
  }
  if(mycpu()->FCFS > 0)
  {
    mycpu()->FCFS--;
    if((mycpu()->FCFS == 0) || (check_no_one_in_queue(2)))
    {
      // cprintf("FCFS time is over %d.\n", mycpu()->FCFS);
      mycpu()->RR = 3;
      yield();
    }
  }
}
```

در واقع هر کدام از 3 متغیر گفته شده مقدار بزرگتر از 0 داشتند از آنها یکی کم میکنیم و اگر 0 شدند به صف بعدی می رویم و یا حتی اگر هر کدام از صف ها پردازه ای درونشان نداشتند ولی هنور زمان داشتند باز هم متغیر آن را 0 کرده و متغیر صف بعدی را مقدار دهی می کنیم.

علاوه بر آن در تابع scheduler هر بار چک میکنیم که کدام یک از این سه مقادیر معتبر هستند و هر موقع باید صف عوض می شد، از حلقه آن سطح خارج شده و وارد حلقه صف بعدی می شویم.

```
//This is for RR
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(mycpu()->RR)
  {
    if(check_no_one_in_queue(0))
    {
      mycpu()->RR = 0;
      mycpu()->SJF = 2;
      break;
    }
    if((p->state != RUNNABLE) || (p->level_queue != 0))
      continue;

    int min_last_exec = find_min_last_exec();
    if(p->last_exec == min_last_exec)
    {
      c->proc = p;
      // cprintf("TICKS: %d PID: %d exec...\n", ticks, p->pid);
      switchuvm(p);
      p->state = RUNNING;

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

      c->proc = 0;
    }
  }
```

تصویر بالا حلقه مربوط به RR است که با توجه به if ابتدایی اگر در صف دیگری باشیم وارد آن نخواهیم شد،علاوه بر آن درون حلقه اگر هر جایی از اجرا دیدیم که دیگر پردازه ای در این سطح نداریم که قابل اجرا باشد(با استفاده از تابع check_no_one_in_queue)سطح اجرا را به SJF تغییر می دهیم و علاوه بر آن از حلقه مربوط به RR، break می کنیم.

```
// SJF
for(int j = 0; j < NPROC; j++){
  if(mycpu()->SJF)
  {
    int count = sort_pcbs_by_burst();
    int seed = (ticks * 17) % 100;
    if(count > 0)
    {
      // cprintf("count is: %d and seed is %d\n", count, seed);
      struct proc *p = sorted_procs[count - 1];
      for (int i = 0; i < count; i++) {
        // cprintf("%d. certainty: %d\n", sorted_procs[i]->pid, sorted_procs[i]->certainty);
          if (sorted_procs[i]->certainty > seed) {
              p = sorted_procs[i];
              // Run the process with the shortest burst time
              break;
          }
      }
      // cprintf("shortest is: %d with burst: %d and certainty: %d\n", p->pid, p->burst, p->certainty);
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    else
    {
      mycpu()->SJF = 0;
      mycpu()->FCFS = 1;
    }
  }
```

تصویر بالا نیز مربوط به SJF است، همانند قسمت قبلی چک می کنیم اگر در این سطح نیستیم وارد آن نمیشویم و همچنین در بدنه حلقه هرگاه دیدیم پردازه فعالی برای اجرا در این سطح نداریم سطح را به FCFS تغییر می دهیم.

```
//FCFS
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(mycpu()->FCFS)
  {
    // cprintf("SJF: %d\n", mycpu()->SJF);
    if(!check_no_one_in_queue(2))
    {
      if((p->state != RUNNABLE) || (p->level_queue != 2))
        continue;
      int first_come = find_first_come();
      if(p->arrival == first_come)
      {
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

        swtch(&(c->scheduler), p->context);
        c->proc = 0;
      }
    }
    else
    {
      mycpu()->FCFS = 0;
      mycpu()->RR = 3;
    }
  }
}
release(&ptable.lock);
```

تصویر بالا نیز مربوط به FCFS است که هرگاه در آن متغیر مربوط به آن در ساختار پردازنده 0 باشد وارد آن نمی شود و همچنین درون آن هرگاه دیدیم پردازه فعالی برای اجرا در آن سطح نداریم متغیر آن را 0 کرده و متغیر مربوط به RR time charing را 3 قرار می دهیم.

توضیحاتی در خصوص موارد اضافه شده:

```
int check_no_one_in_queue(int queue)
{
  struct proc *p;
  int state = 1;
  // acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == RUNNABLE && p->level_queue == queue)
    {
        state = 0;
        break;
    }
  }
  // release(&ptable.lock);
  return state;
}
```

این تابع بررسی می کند که در سطحی که با ورودی اش مشخص می شود پردازه فعالی وجود دارد یا خیر.

```
int find_min_last_exec()
{
  struct proc *p;
  int min = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
      if((p->level_queue == 0) && (p->pid != 0) && (p->state == RUNNABLE))
      {
        min = p->last_exec;
        break;
      }
  }
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
      if((p->level_queue == 0) && (p->pid != 0) && (p->state == RUNNABLE))
      {
        if(p->last_exec < min)
        {
          min = p->last_exec;
        }
      }
  }
  return min;
}
```

این تابع ، پردازه ای را که زمان آخرین اجرایش از بقیه کوچکتر است را یافته و زمان آخرین اجرایش را برمیگرداند.

```
int find_first_come()
{
  struct proc *p;
  int FC = 0;
  int pid = -10;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
      if((p->level_queue == 2) && (p->pid != 0) && (p->state == RUNNABLE))
      {
        FC = p->arrival;
        pid = p->pid;
        break;
      }
  }
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
      if((p->level_queue == 2) && (p->pid != 0) && (p->state == RUNNABLE))
      {
        if(p->arrival < FC)
        {
          FC = p->arrival;
          pid = p->pid;
        }
      }
  }

  cprintf("FIRST COME IS %d BY ARRIVAL %d\n", pid, FC);
  return FC;
}
```

این تابع پردازه ای که زودتر از بقیه وارد شده است را یافته و زمان ورودش را خروجی می دهد و در واقع برای FCFS استفاده می شود.

```
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
  short RR;
  short SJF;
  short FCFS;
};
```

سه مورد آخر بهاطلاعات ساختاری پردازنده اضافه شدند تا هر بار بدانیم کدام سطح را اجرا می کنیم.

```
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)
  int syscall_count;
  int syscall_history[SYSCALL_NUM];
  int level_queue;
  int arrival;
  int last_exec;
  int burst;
  int certainty;
  int wait_cycles;
  int consecutive;
  int queue_arrival;
};
```

Int هایی که در آخر تصویر دیده می شوند به ساختار پردازه اضافه شدند تا در روند scheduling در سطوح مختلف از آنها استفاده کنیم.

الگوریتم کوتاهترین کار، اول:

```
struct proc *sorted_procs[NPROC];
int nextpid = 1;
extern void forkret(void);
extern void trapret(void);

static void wakeup1(void *chan);

int sort_pcbs_by_burst(void) {
    struct proc *p;
    int count = 0;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state == RUNNABLE && p->level_queue == 1) {
            // cprintf("%d is ready\n", p->pid);
            sorted_procs[count++] = p;
        }
    }

    for (int i = 1; i < count; i++) {
        struct proc *key = sorted_procs[i];
        int j = i - 1;
        while (j >= 0 && sorted_procs[j]->burst > key->burst) {
            sorted_procs[j + 1] = sorted_procs[j];
            j--;
        }
        sorted_procs[j + 1] = key;
    }
    return count;
}
```

یک آرایه ثانویه برای ذخیره پردازه هایی که در صف دوم هستند و بر اساس طول اجرا به طور صعودی مرتب شده اند می سازیم و هربار عنصر اول آرایه را برای اجرا انتخاب میکنیم البته مشروط به اینکه سطح اطمینان آن از سطح اطمینانی که داریم بیشتر باشد. همچنین این تابع تعداد پردازه های موجود در صف را پس میدهد.

```
count is: 10 and seed is 51
27. certainty: 21
29. certainty: 25
18. certainty: 52
shortest is: 18 with burst: 9 and certainty: 52
count is: 9 and seed is 68
27. certainty: 21
29. certainty: 25
31. certainty: 37
7. certainty: 21
20. certainty: 20
33. certainty: 57
9. certainty: 45
35. certainty: 85
shortest is: 35 with burst: 11 and certainty: 85
```

همانگونه که در تصویر صفحه بعد واضح است ابتدا آخرین پردازه صف را برای اجرا آماده میکنیم سپس در حلقه بررسی میکنیم آیا پردازه ای دیگر که از آن کوتاهتر است میتواند اجرا شود یا خیر اگر چنین پردازه ای پیدا شد که اجرا میشود در غیر اینصورت پردازه آخر اجرا می شود.

تصویر مقابل هم نمونه ای از اجرا است.

```
for(int j = 0; j < NPROC; j++){
  if(mycpu()->SJF)
  {
    int count = sort_pcbs_by_burst();
    int seed = (ticks * 17) % 100;
    if(count > 0)
    {
      // cprintf("count is: %d and seed is %d\n", count, seed);
      struct proc *p = sorted_procs[count - 1];
      for (int i = 0; i < count; i++) {
        // cprintf("%d. certainty: %d\n", sorted_procs[i]->pid, sorted_procs[i]->certainty);
          if (sorted_procs[i]->certainty > seed) {
              p = sorted_procs[i];
              // Run the process with the shortest burst time
              break;
          }
      }
      // cprintf("shortest is: %d with burst: %d and certainty: %d\n", p->pid, p->burst, p->cert
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    else
    {
      mycpu()->SJF = 0;
      mycpu()->FCFS = 1;
    }
  }
```

ساز و کار افزایش سن:

برای افزایش سن برای هر پردازه در داده ساختار مربوطش یک متغیر اضافه کرده ایم که نشان میدهد از آخرین اجرای آن چقدر میگذرد. با هر تیک تایمر یک واحد به آن اضافه می شود اگر این عدد به 800 برسد یعنی پردازه مزبور 800 تیک است که آماده اجراست ولی فرصت اجرا پیدا نمی کند پس برای جلوگیری از گرسنگی آن را به یک صف بالاتر منتقل میکنیم. برای این منظور از فراخوانی سیستمی دوم استفاده میکنیم. که کد آن در تصویر قابل مشاهده است.

```
void change_queue(int pid , int dest_Q){
  struct proc* p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      if(p->level_queue != dest_Q){
        p->level_queue = dest_Q;
        p->wait_cycles = 0;
        p->queue_arrival = ticks;
      }
      else{
        cprintf("Process is already in queue %d!\n", dest_Q);
      }
      break;
    }
  }
  release(&ptable.lock);
}
```

```c
void handle_aging_and_consecutive()
{
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->level_queue != 0)
    {
      if(p->state == RUNNABLE)
      {
        p->wait_cycles++;
        if(p->wait_cycles == MAX_WAIT)
        {
          cprintf("Process %d moved from src:%d to dest:%d\n", p->pid, p->level_queue, p->level_queue - 1);
          change_queue(p->pid, p->level_queue - 1);
        }
      }
      else if(p->state == RUNNING)
      {
        p->wait_cycles = 0;
        if(p->pid != last_running_pid)
        {
          p->consecutive = 1;
        }
        else
        {
          p->consecutive++;
        }
      }
    }
  }
}
```

این قطعه کد با هر تیک تایمر اجرا میشود تا به تعداد تیک های پردازه هایی که قابل اجرا هستند بیافزاید و همچنین اگر برنامه ای در حال اجرا بود تعداد دفعات اجرای متوالی آن را بروزرسانی کند.

فراخوانی های سیستمی:

فراخوانی سیستمی دوم که در بخش قبل توضیح داده شد و فراخوانی سیستمی اول هم فقط مقادیر مقابل را بروز رسانی میکند.

```c
int burst;
int certainty;
```

فراخوانی سوم هم اطلاعات خواسته شده در صورت پروژه را چاپ میکند که نمونه ای از آن در تصویر زیر قابل مشاهده است.

```
name       pid  state     queue wait       conf      burst   cons    arrival
-----------------------------------------------------------------------------
init       1    SLEEPING  0     0          50        2       0       0
sh         2    SLEEPING  0     0          50        2       0       35
test_RR    3    SLEEPING  2     0          50        2       1       313
test_RR    36   RUNNING   2     0          52        5       1       344
test_RR    37   RUNNING   2     0          21        12       1       345
test_RR    38   RUNNABLE  2     85          92        6       0       345
test_RR    39   RUNNABLE  2     85          65        0       0       345
test_RR    40   RUNNABLE  2     85          40        7       0       345
test_RR    41   RUNNABLE  2     85          17        1       0       345
test_RR    42   RUNNABLE  2     85          96        8       0       345
test_RR    43   RUNNABLE  2     85          77        2       0       345
```