گزارش پروژه سوم

آزمایشگاه سیستم عامل

تاریخ ارسال گزارش: 1404/2/22

محمد مهدی دوست محمدی 810100142
پارسا خوش نظری 810100137
یاسمین اخوین 810100079

فهرست مطالب

پاسخ سوالات: 2
زمان‌بندی چند کلاسی:..... 17
کلاس اول: زودترین موعد 17
کلاس دوم:..... 18
سطح اول: زمان‌بند نوبت گردشی با کوانتوم زمانی 18
سطح دوم: اولین ورود-اولین رسیدگی..... 19
سازوکار افزایش سن در کلاس دوم 20
فراخوانی‌های سیستمی مورد نیاز..... 20
بهینه‌سازی در یافتن پردازه‌ها..... 25
برنامه‌های سطح کاربر 25

پاسخ سوالات:

1) ساختار PCB و همچنن وضعیت های تعریف شده برای هر پردازه را در xv6 پیدا کرده و گزارش کنید. آیا شباهتی میان داده های موجود در ان ساختار و ساختار به تصور کشیده شده در شکل 3.3 منبع درس وجود دارد؟ ذکر حداقل ۵ مورد و معادل آنها در xv6

ساختار منبع درس:

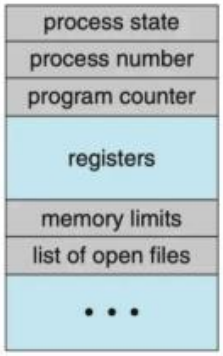

Figure 3.3 Process control block (PCB).

وضعیت پردازش: می‌تواند جدید (new) ، آماده (ready) ، در حال اجرا (running)، منتظر (waiting)، متوقف شده (halted) و غیره باشد.

شمارنده برنامه (Program counter): شمارنده آدرس دستور بعدی که باید برای این پردازش اجرا شود را نشان می‌دهد.

CPU register: تعداد و نوع ثبات‌ها بسته به معماری کامپیوتر متفاوت است. این شامل general-purpose registers، accumulator، index registers، stack pointers به علاوه هر گونه اطلاعات شرطی condition-code می‌باشد. همراه با شمارنده برنامه، این اطلاعات حالت باید هنگام وقوع وقفه (interrupt) ذخیره شوند تا امکان ادامه صحیح پردازش بعد از زمان‌بندی مجدد (reschedule) فراهم شود.

اطلاعات زمان‌بندی (CPU-scheduling information): این اطلاعات شامل اولویت پردازش (process priority) اشاره‌گرها به صف‌های زمان‌بندی (scheduling queues) ، و هر پارامتر زمان‌بندی دیگری می‌باشد.

مدیریت حافظه (Memory-management information): این اطلاعات ممکن است شامل مواردی مانند page tables، segment tables، و base and limit registers باشد، بسته به سیستم عامل مورد استفاده می‌شود.

اطلاعات حسابداری (Accounting information): این اطلاعات شامل مقدار زمان استفاده شده از CPU و زمان، محدودیت‌های زمانی، process numbersو غیره می‌باشد.

I/O status information : این اطلاعات شامل لیست دستگاه‌های I/O تخصیص یافته به پردازش، لیست فایل‌های باز و غیره می‌باشد.

ساختار xv6 :

```
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)
};
```

این ساختار، شامل تمام اطلاعات لازم درباره یک پردازش در xv6 می‌باشد، مانند وضعیت آن، چیدمان حافظه، فایل‌های باز و غیره. که خلاصه‌ای از آن‌ها را بیان می‌کنیم:

- **uint sz** اندازه حافظه پردازش به بایت است. و معادل با **memory limits** است.
- **struct pde_t* pgdir** پوینتر به **page table** ها است و برای مدیریت حافظه مجازی استفاده می‌شود و این بخش هم تا حدی معادل با بخش‌های مربوط به **Memory-Management Information** است.
- **char *kstack** پوینتر به پایین پشته کرنل **process** است.
- **enum procstate state** وضعیت فعلی پردازش مثلاً **UNUSED, RUNNABLE,** و غیره است و معادل با **process state** است.
- **int pid** شناسه پردازش **(Process ID)** است که پردازش را به طور یکتا شناسایی می‌کند و معادل با **number process** است.
- **struct proc *parent** پوینتر به پردازش والد است.

- **struct trapframe *tf** همان **trap frame** کل فعلی است، و برای ذخیره ثبات‌ها استفاده می‌شود و شامل **program counter** هم می‌شود و می‌توان آن را از این لحاظ معادل با **program counter** دانست و از طرف دیگر چون شامل اطلاعات تمام رجیسترها هم می‌شود، میتوان آن را معادل با **cpu registers** دانست.
- **Struct context *context** مورد استفاده توسط **swtch()** برای ذخیره و بازیابی **context** آن پردازش است. و در اصل رجیسترها با مقادیرشان در آن هستند پس و معادل با **registers** است.
- **void *chan** اگر غیر صفر باشد، پردازش روی این کانال **sleeping** است.
- **int killed** اگر غیر صفر باشد، نشان می‌دهد که پردازش به حالت **killed** رفته است.
- **struct file *ofile[NOFILE]** آرایه‌ای از پوینترها به فایل‌های باز است و معادل با **list of open files** است.
- **struct inode *cwd** پوینتری به دایرکتوری فعلی است.
- **char name[16]** نام پردازش است و غالباً برای دیباگ کردن به کار می‌رود.

2) هر کدام از وضعیت‌های تعریف شده معادل کدام وضعیت در شکل 1 می‌باشند؟

وضعیت‌های موجود **در xv6:**

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

- **UNUSED**: در این وضعیت، پردازش هنوز در سیستم جای گرفته نیست، و در حال استفاده نیست. هنوز وارد چرخه اصلی کارکرد سیستم‌عامل نشده و هیچ نقشی در اجرای برنامه‌ها ایفا نمی‌کند. نهایتاً به‌طور حدودی می‌توان آن را معادل با **new** گذاشت چون این وضعیت خانه خالی از جدول پردازه ها است که پردازه ای در آن قرار ندارد.
- **EMBRYO -> new**: پردازه تازه ساخته شده که هنوز آماده اجرا نیست و مقادیر **PCB** والد در **PCB** آن کپی نشده است.
- **SLEEPING -> waiting**: اگر یک پردازه منتظر رخداد خاصی باشد، مانند تکمیل یک عملیات **I/O** یا دریافت یک سیگنال مشخص، در حالت **SLEEPING** به سر می‌برد. در این وضعیت، پردازنده فعالانه کاری انجام نمی‌دهد و منتظر شرایط لازم برای ادامه فعالیتش فراهم شود.
- **RUNNABLE -> ready**: پردازه‌هایی که آماده اجرای روی **CPU** هستند هنوز در صف زمان‌بندی قرار دارند، به عنوان **RUNNABLE** شناخته می‌شوند. این پردازش‌ها تمامی پیش‌نیازهای اجرا را دارند و تنها منتظر اختصاص **CPU** برای شروع فعالیت هستند.
- **RUNNING -> running:** وقتی نوبت یک پردازه برای اجرا روی **CPU** فرا می‌رسد و عملیات محاسباتی یا دستورالعمل‌های برنامه را فعالانه پیش می‌برد، در حالت **RUNNING** قرار دارد.
- **ZOMBIE -> terminated:** پردازه‌ای که اجرایش تمام شده ولی هنوز در جدول پردازه ها وجود دارد چون والد مقدار برگشتی آن را دریافت نکرده است که آن فرزند را **clean** up کند.

3) با توجه به توضیحات گفته شده، کدام یک از توابع موجود در c.proc منجر به انجام گذار از حالت new به حالت ready در xv6 در این گذار از چه حالت/حالت‌هایی به چه حالت/حالت‌هایی تغییر می‌کند؟ پاسخ خود را با پاسخ سوال 2 مقایسه کنید

تابع **fork** در حالت کلی و تابع **userinit** برای اولین پردازه (**initproc**) در ابتدا به کمک تابع **allocproc** پردازه جدیدی از جدول پردازه‌ها که **ptable** است (یعنی آماده‌ی تولید شدن است)، اختصاص می‌دهند و حالت آن **EMBRYO** معادل **new** خواهد شد. شمارنده پردازه ها را یکی افزایش می دهد و برخی مقادیرها را برای پردازه صورت می گیرد و استک آن تولید می شود. در انتها حالت آن را به **RUNNABLE** تغییر می‌دهند. البته که در **userinit**، پردازه از مقادیر اولیه کرده اما در **fork** مقادیر پردازه‌ی **parent** را در **PCB** پردازه‌ی جدید می‌ریزد.

4) سقف تعداد پردازه‌های ممکن در xv6 چه عددی است؟ در صورتی که یک پردازه تعداد زیادی پردازه‌ی فرزند ایجاد کند و از این سقف عبور کند، کرنل چه واکنشی نشان داده و برنامه‌ی سطح کاربر چه بازخوردی دریافت می‌کند؟

سقف تعداد پردازه ها در NPROC برابر با 64 می‌باشد بطور دیفالت:

```
#define NPROC        64  // maximum number of processes
```

اگر تابع **allocproc** هیچ جای خالی‌ای در **ptable** پیدا نکند، به تابع **fork** که آن را صدا زده بود، مقدار صفر بر می‌گرداند و تابع **fork** نیز مقدار 1- را به برنامه سطح کاربر برمی‌گرداند که به معنی ناموفق بودن عملیات **fork** است.

5) چرا نیاز است در ابتدای حلقه تابع scheduler، جدول پردازه‌ها قفل شود؟ آیا در سیستم‌های تک‌پردازه‌ای هم نیاز است این کار صورت بگیرد؟

در سیستم‌عامل**xv6** ، جدول پردازه‌ها (**ptable**) شامل اطلاعات مهمی مانند وضعیت پردازه‌ها است. تابع **scheduler**باید پردازه‌های **RUNNABLE** را از این جدول انتخاب کند. برای جلوگیری از مشکلات همزمانی و تغییرات ناخواسته در وضعیت پردازه‌ها، جدول پردازه‌ها باید قفل شود. دلایل این نیاز در سیستم‌های چندهسته‌ای و تک‌هسته‌ای به شرح زیر است:

1) سیستم‌های چندهسته‌ای: در سیستم‌های چندهسته‌ای، اگر جدول پردازه‌ها قفل نشود، ممکن است دو هسته مختلف به‌طور همزمان پردازه‌ای را برای اجرا انتخاب کنند، که این می‌تواند منجر به استفاده همزمان از منابع مشترک (مثل پشته پردازه‌ها) و خرابی داده‌ها شود. همچنین، قفل کردن جدول پردازه‌ها از مشکلاتی مانند تخصیص نادرست شناسه‌ها جلوگیری کرده و باعث همگام‌سازی بین **exit** و **wait** می‌شود.

2) سیستم‌های تک‌پردازه‌ای: حتی در سیستم‌های تک‌پردازه‌ای، قفل کردن جدول پردازه‌ها ضروری است چون وقفه ها (**interrupts**) و پیش‌گیری (**preemption**) کرنل می‌توانند تغییراتی در وضعیت پردازه‌ها ایجاد کنند. به عنوان مثال، حین اجرای تابع **scheduler** ممکن است وقفه ای رخ دهد و با اجرای **ISR** ، تغییری در وضعیت پردازه‌ها صورت بگیرد پس این حالت هم منجر به ناهمگامی داده‌ها می‌شود.

6) با فرض اینکه xv6 در حالت تک‌هسته‌ای در حال اجراست، اگر یک پردازه به حالت RUNNABLE برود و صف پردازه‌ها در حال طی شدن باشد(proc.c:335) ، در مکانیزم زمان‌بندی xv6 نسبت به موقعیت پردازه در صف، در چه iteration ای امکان schedule پیدا می‌کند؟ (در همان iteration یا در iteration بعدی)

```
  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);

  }
}
```

با توجه به تابع **scheduler**، این تابع در حلقه **for** داخلی به ترتیب روی پردازه های **ptable** جلو می رود. اگر به پردازه ای برسد که استیت **RUNNABLE** داشت آن را اجرا می کند. حال اگر پردازه ای به حالت **RUNNABLE** برود دو حالت داریم. حالت اول اینکه هنوز به این پردازه در حلقه **for** داخلی نرسیدیم، که اگر چنین باشد در ادامه اجرای حلقه به این پردازه می رسیم یا به عبارتی در همان **iteration**، این پردازه امکان **schedule** پیدا می کند. اما اگر در حلقه **for** داخلی، از این پردازه عبور کرده باشیم چون قبلا در حالت **RUNNABLE** نبوده آن را اجرا نکردیم و در ادامه **iteration** دوباره حالت این پردازه را چک نمی کنیم اما در **iteration** بعدی وقتی به این پردازه برسیم، چون در حالت **RUNNABLE** هست آن را اجرا می کنیم. پس در این حالت در **iteration** بعدی این پردازه امکان **schedule** پیدا می کند.

7) رجیسترهای موجود در ساختار context را نام ببرید.

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

(EDI) Extended Destination Index: رجیستر مقصد در عملیات‌های انتقال داده‌ها. معمولاً در عملیات‌های دستکاری رشته‌ها و انتقال داده‌ها به کار می‌رود.

(ESI) Extended Source Index: رجیستر مبدا که به طور عمده در عملیات‌های اشاره‌گر و دستکاری رشته‌ها استفاده می‌شود.

(EBX) Extended Base Register: یک رجیستر عمومی که برای ذخیره‌سازی داده‌ها و مقادیر مختلف در پردازنده استفاده می‌شود.

(EBP) Extended Base Pointer: رجیستر اشاره‌گر پایه که برای مدیریت فریم‌های استک و ارجاع به داده‌ها در استک (مثلاً هنگام فراخوانی توابع) به کار می‌رود.

(EIP) Extended Instruction Pointer: رجیستر اشاره‌گر **instruction** که آدرس **instruction** بعدی که باید اجرا شود را در خود ذخیره می‌کند.

8) همان‌طور که می‌دانید، یکی از مهم‌ترین رجیسترها قبل از هر تعویض متن Program Counter است که نشان می‌دهد روند اجرای برنامه تا کجا پیش رفته است. با ذخیره‌سازی این رجیستر می‌توان محل ادامه برنامه را بازیابی کرد. این رجیستر در ساختار context چه نام دارد؟ این رجیستر چگونه قبل از انجام تعویض متن ذخیره می‌شود؟

رجیستر **EIP (Extended Instruction Pointer)** که در معماری **x86** به آن **PC (Program Counter)** نیز می گویند، نشان‌دهنده پیشرفت اجرای برنامه است و محل دستورالعمل بعدی که باید اجرا شود را ذخیره می‌کند. این رجیستر نشان می‌دهد که پردازنده کجا در برنامه قرار دارد و چه دستورالعملی باید بعداً اجرا شود.

در سیستم‌عامل‌هایی مانند**xv6** ، زمانی که تابع **switch** متن بین پردازش‌ها در **scheduler** یا در حین فراخوانی **yield** اجرا می‌شود، مقدار **EIP** در استک ذخیره می‌شود. این ذخیره‌سازی مانند فراخوانی توابع معمول است که در آن رجیسترها، به ویژه**EIP** ، در استک نگهداری می‌شوند تا وضعیت پردازش جاری حفظ شود.

در حین انجام تعویض متن، مقدار **ESP (Stack Pointer)** به موقعیت جدیدی در استک منتقل می‌شود، جایی که مقدار **EIP**ذخیره شده است. سپس در هنگام سوییچ به پردازش جدید، **EIP**از استک بازیابی شده و پردازش جدید از همان نقطه‌ای که پردازش قبلی متوقف شده بود، ادامه می‌یابد.

این فرآیند ذخیره و بازیابی **EIP** به همراه دیگر رجیسترها در کد اسمبلی موجود در **s.switch** مدیریت می‌شود.

9) همانطور که در قسمت قبل مشاهده کردید، ابتدای تابع scheduler ایجاد وقفه به کمک تابع sti فعال میشود. با توجه به توضیحات این قسمت اگر وقفه ها فعال نمیشد چه مشکلی به وجود می آمد؟

در سیستم‌عامل‌هایی مانند **xv6**، تابع **sti()** در ابتدای هر دور از حلقه ی **scheduler** برای فعال‌سازی وقفه‌ها فراخوانی می‌شود. اگر این وقفه‌ها فعال نشوند، قابلیت **interrupt preemption** که تعویض متن (**context switch**) نمی‌تواند وابسته است، حتی اگر پردازش‌های آماده برای اجرا (**RUNNABLE**) وجود داشته باشند.

یکی از مشکلات جدی‌تری که به وجود می‌آید این است که پردازش‌هایی که منتظر رویدادهای مانند **I/O** هستند، نمی‌توانند این رویدادها را دریافت کنند. برای مثال، اگر پردازشی منتظر ورودی/خروجی **(I/O)** باشد و وقفه‌ها غیرفعال باشند، پردازش هیچ‌گاه قادر به دریافت نتیجه **I/O** نخواهد بود و همچنان در وضعیت **BLOCKED** باقی می‌ماند. این امر می‌تواند منجر به این شود که اگر هیچ پردازش **RUNNABLE** در سیستم نباشد، پردازنده در حالت **idle** باقی بماند و هیچ پردازش جدیدی اجرا نشود، زیرا پردازش‌های موجود در حالت انتظار برای رویدادهای **I/O** با سایر وقفه‌ها خواهند ماند.

در صورتی که وقفه‌ها غیرفعال بمانند، هیچ پردازش جدیدی نمی‌تواند وضعیت **RUNNABLE** تغییر دهد، و در نتیجه هیچ گونه **system call** یا **context switch** انجام شود. به عبارت دیگر، سیستم‌عامل قادر به پاسخ‌گویی به رویدادهای ورودی/خروجی یا تغییر وضعیت پردازش‌ها نخواهد بود

10) به نظر شما وقفه تایمر هر چه مدت یک بار صادر میشود؟

بر اساس داکیومنت **xv6**، در هر ثانیه ۱۰۰ تایمر **interrupt** تولید می کند.

```
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 وقفه تایمر در هر **ms10** یک بار صادر می‌شود.

این نتیجه را میتوان طبق راهنمایی سوال و با افزودن **printf** بعد از هر بار **tick**++ یافت. کافیست مثلا با تابع **cmostime**، زمان آنی را در دو **tick** متوالی بدست آورده و فاصله ی آن ها از هم را بیابیم که البته چون به واحد ثانیه زمان را به ما می دهد، می توان صبر کرد تا فاصله ی دو زمان سنجی، ۱ ثانیه شود و سپس تعداد **tick** ها را پرینت کنیم، که آنگاه تعداد **tick** ها را در یک ثانیه داریم و با میانگین گرفتن از آن، دقیقاً به همین مقدار ۱۰۰ بار در ثانیه می رسیم.

```
ticks++;
if (ticks == 0) {
  cmostime(&starttime);
  cmostime(&currtime);
}
cmostime(&currtime);
if (currtime.second - starttime.second == 1) {
  cprintf("ticks: %d\n", ticks);
  starttime = currtime;
}
```

11) با توجه به توضیحات داده شده، چه تابعی منجر به انجام شدن گذر interrupt در شکل ۱ خواهد شد؟

تابع **yield** پردازنده را پس از یک دوره زمان‌بندی از پردازه‌ای که در حال اجرا است می‌گیرد؛ برای این کار وضعیت پردازه را به **RUNNABLE** تغییر می‌دهد و سپس تابع **sched** را صدا می‌زند که به کمک آن عملیات **context switch** انجام می‌شود.

12) با توجه به توضیحات قسمت scheduler dispatch می‌دانیم زمان‌بندی در xv6 به شکل نوبت گردشی است. حال با توجه به مشاهدات خود در این قسمت، استدلال کنید کوانتوم زمانی این پیاده‌سازی از نوبت گردشی چند میلی ثانیه است؟

در این قسمت از تابع**trap** ، هر بار که وقفه تایمر صادر می‌شود، تابع **yield** زده می‌شود و پردازنده از پردازه‌ی فعلی گرفته می‌شود و به پردازه‌ی بعدی در **ptable** داده می‌شود.

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

چون وقفه تایمر هر ms10 یک بار صادر می‌شود پس کوانتوم زمانی برابر با ms10 ثانیه خواهد شد.

13) تابع wait در نهایت از چه تابعی برای منتظر ماندن برای اتمام کار یک پردازه استفاده می‌کند؟

در تابع **wait** ، پردازه والد منتظر اتمام کار یکی از پردازه‌های فرزند خود می‌شود. این تابع ابتدا از طریق اسکن جدول پردازه‌ها، پردازه‌های فرزند خود را پیدا می‌کند. سپس بررسی می‌کند که آیا پردازه فرزند در وضعیت **ZOMBIE** است

یا خیر. وضعیت **ZOMBIE** به این معنی است که پردازه فرزند به اتمام رسیده است، ولی هنوز منابع آن آزاد نشده‌اند. اگر پردازه فرزند در این وضعیت باشد، تابع **wait** پردازه فرزند را از جدول پردازه‌ها حذف کرده، منابع آن را آزاد می‌کند و سپس **pid** پردازه فرزند را به پردازه والد باز می‌گرداند.

اگر پردازه‌ای در وضعیت **ZOMBIE** پیدا نشود، به این معناست که پردازه‌های فرزند هنوز به اتمام نرسیده‌اند. در این صورت، پردازه والد باید منتظر بماند تا یکی از پردازه‌های فرزند به پایان برسد. برای این کار، تابع **sleep** از تابع **wait** استفاده می‌کند. تابع **sleep** پردازه والد را به حالت خواب (**sleep**) می‌برد تا زمانی که یکی از پردازه‌های فرزند به وضعیت **ZOMBIE** برسد و سپس پردازه والد بتواند از خواب بیدار شود و اطلاعات پردازه فرزند را دریافت کند.

تابع **sleep** در اینجا با دو آرگومان فراخوانی می‌شود:

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

که در آن:

- **curproc:** اشاره‌گر به پردازه والد است که منتظر می‌ماند.
- **&ptable.lock:** قفل جدول پردازه‌ها است که باعث می‌شود پردازه والد به هنگام انتظار (**sleep**) نتواند دسترسی به جدول پردازه‌ها را تغییر دهد تا از مشکلات همزمانی جلوگیری شود.

با این کار، پردازه والد تا زمانی که یکی از پردازه‌های فرزند به وضعیت **ZOMBIE** برسد و منابع آن آزاد شود، به حالت خواب می‌رود. هنگامی که پردازه فرزند اتمام کار می‌کند، از طریق تابع **wakeup1** در تابع**proc_exit** ، پردازه والد از حالت خواب بیدار می‌شود و ادامه عملیات خود را انجام می‌دهد.

14) با توجه به پاسخ سوال قبل، استفاده‌های دیگر این تابع چیست؟ (ذکر یک نمونه)

تابع sleep در xv6 برای مسدود کردن پردازه‌ها تا زمان وقوع یک رویداد خاص یا برآورده شدن یک شرط استفاده می‌شود. علاوه بر کاربرد در تابع wait برای منتظر ماندن پردازه والد تا اتمام پردازه فرزند، این تابع در موارد زیر نیز کاربرد دارد:

- انتظار برای تایمر/تاخیر :پردازه‌ها می‌توانند برای مدت مشخصی به خواب بروند، مانند استفاده از sys_sleep برای ایجاد تاخیر زمانی.

```
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(myproc()->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```

همگام‌سازی در درایورهای دستگاه: پردازه‌ها در حین انتظار برای عملیاتI/O ، مانند دریافت ورودی از کاربر در درایور کنسول، به خواب می‌روند.

```
consoleread(struct inode *ip, char *dst, int n)
{
  uint target;
  int c;

  iunlock(ip);
  target = n;
  acquire(&cons.lock);
  while(n > 0){
    while(input.r == input.w){
      if(myproc()->killed){
        release(&cons.lock);
        ilock(ip);
        return -1;
      }
      sleep(&input.r, &cons.lock);
```

- همگام‌سازی پردازش‌ها با رویدادها: پردازه‌ها هنگام انتظار برای یک سری رویدادها مثل نوشته شدن یا خوانده شدن داده از منابعی مانند لوله‌ها (pipe)، به خواب می‌روند.

15) با این تفاسیر چه تابعی در سطح کرنل، منجر به آگاه سازی پردازه از رویدادی است که برای آن منتظر بوده است؟

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

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
      if(p->readopen == 0 || myproc()->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}
```

در سیستم‌عامل **xv6**، تابع **wakeup** برای بیدار کردن پردازه‌هایی استفاده می‌شود که منتظر یک رویداد خاص بوده‌اند. این تابع ابتدا قفل جدول پردازه‌ها (**ptable.lock**) را می‌گیرد و سپس تابع **wakeup1** را صدا می‌زند تا پردازه‌های منتظر را بیدار کند.

```
// 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;
}

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

تابع **wakeup1** در واقع جدول پردازه‌ها را پیمایش کرده و پردازه‌ها را از حالت **SLEEPING** به **RUNNABLE** تغییر وضعیت می‌دهد، تنها در صورتی که پردازه منتظر همان رویدادی باشد که در **chan** مشخص شده است. این بررسی با مقایسه متغیر **chan** پردازه با داده‌ای که به **wakeup1** ارسال شده انجام می‌شود.

هر پردازه‌ای می‌تواند با استفاده از **wakeup** پردازه دیگر را بیدار کند و در کل تابع **wakeup** باعث می‌شود پردازه‌هایی که منتظر رویدادی خاص بودند، پس از وقوع آن رویداد، به وضعیت **RUNNABLE** تغییر کنند و آماده اجرا شوند.

16) با توجه به پاسخ سوال ۹، این تابع منجر به گذر از چه وضعیتی به چه وضعیتی در شکل ۱ خواهد شد؟

تابع **wakeup()** باعث تغییر وضعیت یک پردازه از **SLEEPING** منتظر رویداد یا همان **waiting** در منبع درس به **RUNNABLE** آماده برای زمان‌بندی یا همان **ready** در منبع درس می‌شود.

17) آیا تابع دیگری وجود دارد که منجر به انجام این گذر شود؟ نام ببرید.

در سیستم‌عامل**xv6** ، علاوه بر تابع **wakeup()** که پردازه‌ها را از وضعیت **SLEEPING** به **RUNNABLE** تغییر می‌دهد، تابع **kill()** نیز می‌تواند این گذر را انجام دهد. وقتی پردازه‌ای در وضعیت خواب (**SLEEPING**) باشد و سیگنال خاتمه دریافت کند، تابع **kill()** وضعیت آن را به **RUNNABLE** تغییر می‌دهد تا پردازه قادر به پاسخ دادن به سیگنال خاتمه باشد.

کد تابع kill:

```
// 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;
}
```

سایر توابع:

- **wakeup()**: پردازه‌هایی که منتظر یک رویداد خاص هستند را بیدار می‌کند و وضعیت آن‌ها را به **RUNNABLE**تغییر می‌دهد.
- **exit()**: هنگام خروج پردازه‌ها، پردازه‌های منتظر بیدار شده و به **RUNNABLE** تغییر وضعیت می‌دهند.
- **I/O Operations**: پردازه‌هایی که منتظر عملیات **I/O** هستند، پس از اتمام عملیات، بیدار شده و وضعیت آن‌ها به **RUNNABLE** تغییر می‌کند.

18)در بخش ۳.۲.۲ منبع درس با پردازه‌های Orphan آشنا شدید، رویکرد xv6 در رابطه با این گونه پردازه‌ها چیست؟

پردازه‌هایی که اجرای والد آن‌ها به هر دلیلی زودتر از پردازه فرزند تمام می‌شود، پردازه یتیم یا **Orphan** نامیده می‌شوند. در **xv6**، پردازه‌های فرزند زمانی که تمام می‌شوند به حالت **ZOMBIE** می‌روند تا توسط والد خود اطلاعاتشان جمع‌آوری شوند. اما اگر یک پردازه یتیم شده باشد، والدی وجود ندارد که اطلاعات این پردازه را جمع‌آوری کند. در **xv6** زمانی که یک پردازه به حالت **ZOMBIE** می‌رسد ولی والد آن موجود نیست، به پردازه **initproc** که همواره در حال اجرا است ارجاع داده می‌شود و اطلاعات آن توسط پردازه **initproc** جمع‌آوری می‌شود و این فرزند از

ptable، در فراخوانی تابع **wait** توسط **initproc** پاک می‌شود. این روند باعث می‌شود که پردازه‌های یتیم منابع را الکی اشغال نکنند.

```
// 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);
  }
}
```

19) فرض کنید در این مکانیزم زمان‌بندی، تمامی پردازنده‌ها مشغول اجرای پردازه‌های کلاس اول باشند. در این صورت برای shell xv6 چه اتفاقی می‌افتد؟ دیدگاه خود را نیز نسبت به این موضوع شرح دهید.

در این حالت، چون **shell** در کلاس دوم قرار دارد و اولویت پایین‌تری نسبت به کلاس اول دارد، هیچگاه توسط زمان‌بند انتخاب نخواهد شد. بنابراین **shell** اجرا نمی‌شود و کاربر نمی‌تواند هیچ فرمانی وارد کند یا برنامه‌ای را اجرا کند. این منجر به قفل شدن سیستم از دید کاربر می‌شود و امکان تعامل با سیستم از بین می‌رود.

20) مقدار CPUS را مجددا به عدد 2 برگردانید. آیا همچنان ترتیبی که ثبلا مشاهده میکردید پا برجاست؟ علت این امر چیست؟

با افزودن تعداد پردازنده ، چون هر دو پردازنده با گرفتن قفل ptable وارد بخش حیاتی scheduler() شده و پردازه ای که طبق سیاست های زمانبندی انتخاب می شود را برای اجرا برمیدارد، پردازنده دیگر نیز هر گاه قفل ptable آزاد شود همین کار را میکند و هر دو از روی یک صف ptable بر اساس سیاست های تعیین شده پردازه برمیدارند.

21) در پیاده‌سازی زمان‌بندی EDF به دلیل ماهیت Soft Real-Time بودن اینگونه تسک‌ها از فرض پیاده‌سازی مطرح شده استفاده کردیم. در صورتی که اینگونه تسک‌ها Hard Real-Time بودند، به نظر شما استفاده از چه فرض پیاده‌سازی مناسب‌تر می‌بود؟

در صورت **Hard Real-Time** بودن تسک‌ها، الگوریتم **EDF** نمی‌تواند تضمین دهد که تسک‌ها حتماً قبل از موعد خود اجرا شوند. برای چنین شرایطی باید از الگوریتم‌هایی با تضمین زمانی سخت‌تر مانند **Rate Monotonic Scheduling (RMS)** استفاده شود. **EDF** برای **Soft Real-Time** مناسب‌تر است زیرا احتمال از دست رفتن موعد در آن وجود دارد و سیستم در برابر آن مقاوم است.

22) چرا به علت مدت زمانی که پردازه در وضعیت SLEEPING می‌باشد به عنوان انتظار پردازه از نظر زمان‌بندی در نظر گرفته نمی‌شود؟

در واقع زمان انتظار برای یک پردازه نشان دهنده مدت زمانی است که این پردازه آماده اجرا است ولی منتظر این است که روی **cpu** اجرا شود؛ به عبارتی در حالت **RUNNABLE** قرار دارد اما **cpu** ای به آن اختصاص داده نشده است. اما زمانی که پردازه در حالت **SLEEPING** است یعنی منتظر یک عامل خارجی مثل عملیات **I/O** یا وقفه های انتظار است تا از این حالت در بیاید و در این مدت منتظر این نیست که **cpu** به آن اختصاص داده شود تا اجرا شود پس در رقابت برای گرفتن **cpu** نیست. پس منطقی نیست که زمان انتظار آن افزایش یابد. همچنین در الگوریتم **aging**، هدف این است که از **starvation** پردازه‌های **CPU-bound** جلوگیری کنیم. اگر پردازه‌های **SLEEPING** را در افزایش زمان انتظار دخیل کنیم، صف آن‌ها با اینکه در انتظار **cpu** نیستند تغییر می‌کند و این باعث می‌شود پردازه‌های **CPU-bound** مدت زیادی منتظر **cpu** بمانند و همچنان فرصت اجرای کمتری داشته باشند (چون اولویت پردازه‌های **SLEEPING** هم افزایش پیدا کرده است).

23) با توجه به دانشی که از پاسخ به سوالات قبل به دست آورده‌اید و پیاده‌سازی که تا به اینجا انجام داده‌اید، در چه توابعی می‌بایست از مقادیر مربوط به تعداد پردازه‌های آماده هر صف کم و یا به آن اضافه کنیم؟ (می‌توانید از شکل ۱ برای مرور فرایند طی شده نیز کمک بگیرید)

برای پیدا کردن تعداد پردازه‌های **RUNNABLE** در هر صف:

- باید در ساختار صف‌ها متغیرهایی برای شمارش پردازه‌ها اضافه شود.
- در توابعی مثل **enqueue, dequeue, scheduler, sleep, wake,** و... که وضعیت پردازه‌ها تغییر می‌دهند، مقدار این شمارنده‌ها باید به‌روزرسانی شود.
- همچنین در هر تغییر صف) مانند بالا رفتن سطح پردازه به خاطر (**aging** نیز باید این مقادیر اصلاح شوند.
- هنگام فراخوانی سیستمی نیز می‌توان این شمارنده‌ها را چاپ کرد.

زمان‌بندی چند کلاسی:

سیاست های لازم برای زمانبندی در تابع schedule() در فایل proc.c :

کلاس اول: زودترین موعد در مود Soft Real Time

```
// ---------- Class 1: EDF ----------
if (count_edf > 0)
{
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == RUNNABLE && p->sched_class == 1)
    {
      if (chosen == 0 || p->deadline < chosen->deadline)
        chosen = p;
    }
  }
}
```

شمارنده های count_fcfs , count_edf , count_rr جهت بهینه سازی پردازنده بکار برده شده اند. این شمارنده ها مانع از جست و جوی بیهوده به دنبال پردازه های که وجود ندارند میشود.

کلاس دوم:

سطح اول: زمان‌بند نوبت گردشی با کوانتوم زمانی

```
else if (count_rr > 0)
{
  // ---------- Class 2, Level 1: RR ----------
  if (chosen == 0)
  {
  // Loop over process table looking for process to run.
  p = nextrrp;
  do
  {
    if (p->state != RUNNABLE || p->sched_level != 1)
    {
      if (++p == &ptable.proc[NPROC])
        p = ptable.proc;
      continue;
    }

    chosen = p;

    if (++p == &ptable.proc[NPROC])
      p = ptable.proc;
    nextrrp = p;
    break;
  } while (p != nextrrp);

  }
}
```

سطح دوم: اولین ورود-اولین رسیدگی

```
// ---------- Class 2, Level 2: FCFS ----------
if (chosen == 0)
{
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == RUNNABLE && p->sched_class == 2 && p->sched_level == 2)
    {
      p->waited_ticks++;
      if (chosen == 0 || p->arrival_time < chosen->arrival_time)
        chosen = p;
    }
  }
}
}
```

پس از اعمال سه سیاست فوق ، نهایتا طی اجرای قطعه کد زیر، پردازه برای اجرا به پردازنده سپرده می شود.

```
// ---------- Run Chosen Process ----------
if (chosen)
{
  c->proc = chosen;
  switchuvm(chosen);
  chosen->state = RUNNING;

  // Decrement count before switching
  if (chosen->sched_class == 1)
    count_edf--;
  else if (chosen->sched_class == 2 && chosen->sched_level == 1)
    count_rr--;
  else if (chosen->sched_class == 2 && chosen->sched_level == 2)
    count_fcfs--;
  chosen->consecutive_run = 0; // reset before a fresh run
  chosen->rr_ticks = 0;
  chosen->waited_ticks = 0;
  acquire(&tickslock);
  chosen->last_run = ticks;
  release(&tickslock);

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

  c->proc = 0;
}

release(&ptable.lock);
}

}
```

سازوکار افزایش سن در کلاس دوم

برای این منظور از سیستم کال هایی استفاده شده است که در ادامه هر یک توضیح داده می شوند.

این سیستم کال ها در تابع trap.c:trap() ، هر گاه وقفه ی تایمر سر برسد و پردازه سطح دومی ، بیش از 800 تیک منتظر مانده باشد صدا میشود:

```
switch (tf->trapno)
{
case T_IRQ0 + IRQ_TIMER:
  int osTicks = 0;
  if (cpuid() == 0)
  {
    acquire(&tickslock);
    ticks++;
    osTicks = ticks;
    wakeup(&ticks);
    release(&tickslock);
    update_wait_time(osTicks); // Accumulate waited_ticks for all RUNNABLE processes
  }
```

در تصویر بالا مشخص است که با هر بار سر آمدن وقفه ی تایمر، پارامتر های زمانی پردازه ها بروز می شوند.

```
int update_wait_time(int osTicks)
{
  acquire(&ptable.lock);
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == RUNNABLE)
    {
      p->waited_ticks = osTicks - p->arrival_time;
      if (p->sched_class == 2 && p->sched_level == 2)
      {
        if (p->waited_ticks >= AGING_THRESHOLD)
        {
          release(&ptable.lock);
          change_sched_level(p->pid, 1);
          acquire(&ptable.lock);
          p->arrival_time = osTicks;
        }
      }
    }
  }
  release(&ptable.lock);
  return 0;
}
```

سیاست افزایش سن در سیستم کال فوق مشاهده می شود.

فراخوانی‌های سیستمی مورد نیاز

1- تعویض سطح زمانبندی

```
int change_sched_level(int pid, int target_level)
{
  if (target_level != 1 && target_level != 2)
  {
    cprintf("Invalid target level!");
    return -1;
  }

  acquire(&ptable.lock);
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      if (p->sched_level == target_level)
      {
        cprintf("Process is already in target level!");
        return -1;
      }
      if (p->state == RUNNABLE)
      {
        if (p->sched_level == 1)
        {
          count_rr--;
          count_fcfs++;
        }
        else
        {
          count_fcfs--;
          count_rr++;
        }
      }
      cprintf("PID %d : Level %d to %d\n", pid, p->sched_level,
target_level);
      p->sched_level = (target_level == 2) ? 2 : 1;
      release(&ptable.lock);
      return 0;
    }
  }
```

```
    release(&ptable.lock);
    return -1;
  }
```

2- چاپ اطلاعات پردازه ها

```
int print_sched_info(void)
{
  static char *states[] = {
      [UNUSED] "UNUSED",
      [EMBRYO] "EMBRYO",
      [SLEEPING] "SLEEPING",
      [RUNNABLE] "RUNNABLE",
      [RUNNING] "RUNNING",
      [ZOMBIE] "ZOMBIE"};
  acquire(&tickslock);
  cprintf("----------------------\n"
          "Tick: %d \n" , ticks);
  release(&tickslock);
  static int columns[] = {16, 8, 12, 12, 12, 12, 12, 17, 9, 10, 13};
  cprintf("-----------------------------------------------------------------------
-------------------------------------------------------------\n"
          "Process_Name    PID     State       Class       Algorithm   Wait_time
  Deadline    Consecutive_run  Arrival  RR_time   Running_Time \n");

  struct proc *p;
  acquire(&ptable.lock);

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

    const char *state;
    if (p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";

    char *algorithm = "FCFS";
    if (p->sched_class == 1)
      algorithm = "EDF";
    else if (p->sched_level == 1)
      algorithm = "RR";
```

```c
    char *class = "Normal";
    if (p->sched_class == 1)
      class = "Real-Time";

    // Print Name
    cprintf("%s", p->name);
    printspaces(columns[0] - strlen(p->name));
    // Print PID
    cprintf("%d", p->pid);
    printspaces(columns[1] - digitcount(p->pid));
    // Print State
    cprintf("%s", state);
    printspaces(columns[2] - strlen(state));
    // Print Class
    cprintf("%s", class);
    printspaces(columns[3] - strlen(class));
    // Print Algorithm
    cprintf("%s", algorithm);
    printspaces(columns[4] - strlen(algorithm));
    // Print Wait time
    cprintf("%d", p->waited_ticks);
    printspaces(columns[5] - digitcount(p->waited_ticks));
    // Print Deadline
    cprintf("%d", p->deadline);
    printspaces(columns[6] - digitcount(p->deadline));
    // Print Consecutive run
    cprintf("%d", p->max_consecutive_run);
    printspaces(columns[7] - digitcount(p->max_consecutive_run));
    // Print Arrival
    cprintf("%d", p->arrival_time);
    printspaces(columns[8] - digitcount(p->arrival_time));
    // Print spent time in RR
    cprintf("%d", p->rr_ticks);
    printspaces(columns[9] - digitcount(p->rr_ticks));
    // Print whole running time
    cprintf("%d", p->runnig_time);
    printspaces(columns[10] - digitcount(p->runnig_time));

    cprintf("\n");
  }
  release(&ptable.lock);
  return 0;
}
```

3- ساخت پردازه با موعد از پیش تعیین شده

```
int set_deadline_for_process(int pid, int deadline)
{
  if (deadline < 0)
  {
    cprintf("Invalid deadline!");
    return -1;
  }

  acquire(&ptable.lock);
  struct proc *p;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      if (p->state == RUNNABLE)
      {
        if (p->sched_level == 1)
        {
          count_rr--;
          count_edf++;
        }
        else if (p->sched_level == 2)
        {
          count_fcfs--;
          count_edf++;
        }
      }
      p->sched_level = 0;
      p->sched_class = 1;
      p->deadline = deadline;
      cprintf("PID %d : Deadline = %d \n", pid, p->deadline);

      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

پس از فورک شدن یک پردازه، پردازه مادر و یا خود پردازه با صدا زدن این فراخوانی سیستمی اولویت فرزند را بالا میبرد

بهینه‌سازی در یافتن پردازه‌ها

برای بهینه سازی در یافتن پردازه همانطور که قبلا توضیح داده شد شمارنده هایی تعریف و صفر نبودنشان بعنوان شرط جستجو در ptable در نظر گرفته شده است.

```
int count_edf = 0;
int count_rr = 0;
int count_fcfs = 0;
```

برنامه‌های سطح کاربر

1- برنامه سطح کاربر schedtest برای آزمایش همه ی زمانبندی ها در نظر گرفته شده است.

```
int main(void)
{
    int pid;

    for (int i = 0; i < NPROC; i++)
    {
        pid = fork();
        if (pid < 0)
        {
            printf(1, "scheduletest: fork failed\n");
            exit();
        }
        if (pid == 0)
        {
            printf(1, "scheduletest: starting process %d\n", getpid());
            busywork(5, getpid());
            print_sched_info(); // Custom syscall
            exit();
        }

        if (pid % 3 == 0)
            set_deadline_for_process(pid, 400 + i*10 );

        if (pid % 4 == 0 && pid % 3 != 0)
            change_sched_level(pid, 1);
```

```
        print_sched_info();
    }
    for (int j = 0; j < 50; j++)
        busywork(1, 8008);

    for (int i = 0; i < NPROC; i++)
    {
        printf(1, "scheduletest: ending process %d\n", wait());
        print_sched_info();
    }

    exit();
}
```

در این برنامه یک پردازنده مادر 10 پردازه را فورک میکند و به هر یک اولویت خاصی می دهد. خروجی این برنامه صحت عملکرد زمانبندی را نشان میدهد.

```
$ schedtest
PID 4 : Level 2 to 1
----------------------
Tick: 558
-----------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  1                2        1         1
sh              2       SLEEPING    Normal      RR          0           1000000000  0                557      0         0
schedtest       3       RUNNING     Normal      FCFS        0           1000000000  1                557      0         1
schedtest       4       RUNNABLE    Normal      RR          0           1000000000  0                558      0         0
scheduletest: starting process 4
RR proc 4  yielded, RR ticks: 3
RR proc 4  yielded, RR ticks: 3
RR proc 4  yielded, RR ticks: 3
RR proc 4  yielded, RR ticks: 3
```

پردازه فورک شده با آیدی 4 در ابتدا به سطح اول می رود و به همین علت سریعا با سیاست RoundRobin اجرا می شود. مشاهده می شود که در هر کوانتوم زمانی که این پردازه از پردازنده گرفته می شود، دوباره همین پردازه اجرا می شود چون پردازه دیگری در این سطح نداریم.

```
RR proc 4  yielded, RR ticks: 3
RR proc 4  yielded, RR ticks: 3
RR proc 4  yielded, RR ticks: 3
RR proc 4  yielded, RR ticks: 3
RR proc 4  yielded, RR ticks: 3
fib: 832040 -   busyjob 4 finished.
----------------------
Tick: 790
-----------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  1                2        1         1
sh              2       SLEEPING    Normal      RR          0           1000000000  0                557      0         0
schedtest       3       RUNNABLE    Normal      FCFS        232         1000000000  2                559      0         2
schedtest       4       RUNNING     Normal      RR          0           1000000000  3                790      1         232
----------------------
Tick: 792
-----------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  1                2        1         1
sh              2       SLEEPING    Normal      RR          0           1000000000  0                557      0         0
schedtest       3       RUNNING     Normal      FCFS        0           1000000000  2                792      0         3
schedtest       4       ZOMBIE      Normal      RR          0           1000000000  3                790      2         233
schedtest       5       RUNNABLE    Normal      FCFS        1           1000000000  0                792      0         0
PID 6 : Deadline = 420
----------------------
```

پس از پایان این پردازه ، گزارش وضعیت پردازه ها چاپ می شود و سپس پردازه های بعدی فورک می شوند. در تصویر بالا مشاهده می شود که پس از اتمام پردازه 4 و فورک شدن پردازه 5، پردازه 6 فورک شده و برایش ددلاین تعیین شده است.

```
PID 6 : Deadline = 420
-----------------------
Tick: 794
-----------------------------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
scheduletest: starting process 6
fib: 832040 -   busyjob 6 finished.
-----------------------
Tick: 1019
-----------------------------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  1                2        1         1
sh              2       SLEEPING    Normal      RR          0           1000000000  0                557      0         0
schedtest       3       RUNNABLE    Normal      FCFS        225         1000000000  3                795      0         5
schedtest       4       ZOMBIE      Normal      RR          0           1000000000  3                790      2         233
schedtest       5       RUNNABLE    Normal      FCFS        228         1000000000  0                792      0         0
schedtest       6       RUNNING     Real-Time   EDF         0           420         225              794      0         225
scheduletest: starting process 5
fib: 832040 -   busyjob 5 finished.
-----------------------
Tick: 1237
-----------------------------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  1                2        1         1
```

در تصویر بالا مشاهده می شود که پردازه 6 زودتر از 5 اجرا شده و پایان یافته است و سپس پردازه 5 شروع به کار کرده است.

خروجی به همین مموال پیش می رود . در انتها پردازه 13 ایجاد می شود که هم سطح با پردازه مادر است و پردازه مادر وارد یک اجرای طولانی(busyjob 8008) می شود.

```
-----------------------
Tick: 2556
-----------------------------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  1                2        1         1
sh              2       SLEEPING    Normal      RR          0           1000000000  0                557      0         0
schedtest       3       RUNNING     Normal      FCFS        0           1000000000  6                2555     0         19
schedtest       4       ZOMBIE      Normal      RR          0           1000000000  3                790      2         233
schedtest       5       ZOMBIE      Normal      FCFS        0           1000000000  217              792      0         217
schedtest       6       ZOMBIE      Real-Time   EDF         0           420         226              794      0         226
schedtest       7       ZOMBIE      Normal      FCFS        0           1000000000  211              1239     0         211
schedtest       8       ZOMBIE      Normal      RR          0           1000000000  3                1467     0         225
schedtest       9       ZOMBIE      Real-Time   EDF         0           450         214              1679     0         214
schedtest       10      ZOMBIE      Normal      FCFS        0           1000000000  213              1895     0         213
schedtest       11      ZOMBIE      Normal      FCFS        0           1000000000  230              1898     0         230
schedtest       12      ZOMBIE      Real-Time   EDF         0           480         212              1899     0         212
schedtest       13      RUNNABLE    Normal      FCFS        1           1000000000  0                2556     0         0
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
```

این کار آن قدر طول می کشد تا سیستم عامل مجبور شود برای جلوگیری از گرسنگی پردازه 13 ، آنرا به سطح بالاتر بیاورد:

```
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
PID 13 : Level 2 to 1
scheduletest: starting process 13
RR proc 13  yielded, RR ticks: 3
RR proc 13  yielded, RR ticks: 3
RR proc 13  yielded, RR ticks: 3
RR proc 13  yielded, RR ticks: 3
RR proc 13  yielded, RR ticks: 3
RR proc 13  yielded, RR ticks: 3
```

پردازه 13 بدون رقیب شروع به اجرا میکند و پس از پایان ، پردازه مادر به کار خود ادامه می دهد .

```
RR proc 13  yielded, RR ticks: 3
RR proc 13  yielded, RR ticks: 3
RR proc 13  yielded, RR ticks: 3
fib: 832040 -   busyjob 13 finished.
-----------------------
Tick: 3584
-----------------------------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  1                2        1         1
sh              2       SLEEPING    Normal      RR          0           1000000000  0                557      0         0
schedtest       3       RUNNABLE    Normal      FCFS        229         1000000000  801              3356     0         818
schedtest       4       ZOMBIE      Normal      RR          0           1000000000  3                790      2         233
schedtest       5       ZOMBIE      Normal      FCFS        0           1000000000  217              792      0         217
schedtest       6       ZOMBIE      Real-Time   EDF         0           420         226              794      0         226
schedtest       7       ZOMBIE      Normal      FCFS        0           1000000000  211              1239     0         211
schedtest       8       ZOMBIE      Normal      RR          0           1000000000  3                1467     0         225
schedtest       9       ZOMBIE      Real-Time   EDF         0           450         214              1679     0         214
schedtest       10      ZOMBIE      Normal      FCFS        0           1000000000  213              1895     0         213
schedtest       11      ZOMBIE      Normal      FCFS        0           1000000000  230              1898     0         230
schedtest       12      ZOMBIE      Real-Time   EDF         0           480         212              1899     0         212
schedtest       13      RUNNING     Normal      RR          0           1000000000  3                3584     1         229
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
fib: 832040 -   busyjob 8008 finished.
```

نهایتا گزارش کاملی از اجرای تمام پردازه ها قابل مشاهده است:

```
scheduletest: ending process 4
----------------------
Tick: 4993
-----------------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  1                2        1         1
sh              2       SLEEPING    Normal      RR          0           1000000000  0                557      0         0
schedtest       3       RUNNING     Normal      FCFS        0           1000000000  1408             3586     0         2226
schedtest       5       ZOMBIE      Normal      FCFS        0           1000000000  217              792      0         217
schedtest       6       ZOMBIE      Real-Time   EDF         0           420         226              794      0         226
schedtest       7       ZOMBIE      Normal      FCFS        0           1000000000  211              1239     0         211
schedtest       8       ZOMBIE      Normal      RR          0           1000000000  3                1467     0         225
schedtest       9       ZOMBIE      Real-Time   EDF         0           450         214              1679     0         214
schedtest       10      ZOMBIE      Normal      FCFS        0           1000000000  213              1895     0         213
schedtest       11      ZOMBIE      Normal      FCFS        0           1000000000  230              1898     0         230
schedtest       12      ZOMBIE      Real-Time   EDF         0           480         212              1899     0         212
schedtest       13      ZOMBIE      Normal      RR          0           1000000000  3                3584     2         230
```

در تصویر فوق پردازه 4 پایان یافته و exit کرده به همین دلیل چاپ نشده.

در تصویر فوق علاوه بر ددلاین و بیشترین زمان اجرای پشت سر هم، کل زمان اجرای یک پردازه و زمان آخرین اجرای گردشی(Round Robin) نیز آورده شده است.

2- برنامه سطح کاربر rrtest جهت اطمینان از عملکرد درست پردازه های سطح اول از کلاس دوم طراحی شده. به طوری که 5 پردازه همزمان فورک شده و سطحشان به 1 تغییر می کند تا در سطح اول با هم رقابت کنند.

```c
int main()
{
    int pid;

    for (int i = 0; i < N; i++)
    {
        pid = fork();
        if (pid == 0)
        {
            busywork(5,getpid());
            print_sched_info(); // Watch RR rotation, quantum, etc.
            exit();
        }
        change_sched_level(pid, 1); // Level 1 = RR
    }

    for (int i = 0; i < N; i++)
        wait();

    busywork(10 , 8008);
    print_sched_info();
    printf(1, "RR test complete\n");
    exit();
}
```

صحت زمانبندی در تصویر زیر قابل مشاهده است:

```
$ rrtest
PID 4 : Level 2 to 1
PID 5 : Level 2 to 1
PID 6 : Level 2 to 1
PID 7 : Level 2 to 1
PID 8 : Level 2 to 1
RR proc 4  yielded, RR ticks: 3
RR proc 5  yielded, RR ticks: 3
RR proc 6  yielded, RR ticks: 3
RR proc 7  yielded, RR ticks: 3
RR proc 8  yielded, RR ticks: 3
RR proc 4  yielded, RR ticks: 3
RR proc 5  yielded, RR ticks: 3
RR proc 6  yielded, RR ticks: 3
RR proc 7  yielded, RR ticks: 3
RR proc 8  yielded, RR ticks: 3
RR proc 4  yielded, RR ticks: 3
RR proc 5  yielded, RR ticks: 3
RR proc 6  yielded, RR ticks: 3
RR proc 7  yielded, RR ticks: 3
RR proc 8  yielded, RR ticks: 3
RR proc 4  yielded, RR ticks: 3
RR proc 5  yielded, RR ticks: 3
RR proc 6  yielded, RR ticks: 3
RR proc 7  yielded, RR ticks: 3
RR proc 8  yielded, RR ticks: 3
```

مشاهده می شود که هر پردازه بعد از یک کوانتوم زمانی (3 تیک = 30 میلی ثانیه) متوقف شده و پردازه دیگری اجرا می شود .

در ادامه پردازه ها با همان ترتیبی ک شروع به اجرا کرده بودند به اتمام می رسند:

```
  busyjob 4 finished.
-----------------------
Tick: 349
RR proc 4  yielded, RR ticks: 3
  busyjob 5 finished.
-----------------------
Tick: 351
-----------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  0                2        0         0
sh              2       SLEEPING    Normal      RR          0           1000000000  0                286      0         0
rrtest          3       SLEEPING    Normal      FCFS        0           1000000000  0                287      0         0
rrtest          4       RUNNABLE    Normal      RR          1           1000000000  3                350      3         15
rrtest          5       RUNNING     Normal      RR          0           1000000000  3                338      1         13
rrtest          6       RUNNABLE    Normal      RR          10          1000000000  3                341      3         12
rrtest          7       RUNNABLE    Normal      RR          7           1000000000  3                344      3         12
rrtest          8       RUNNABLE    Normal      RR          4           1000000000  3                347      3         12
  busyjob 6 finished.
-----------------------
Tick: 353
-----------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  0                2        0         0
sh              2       SLEEPING    Normal      RR          0           1000000000  0                286      0         0
rrtest          3       RUNNABLE    Normal      FCFS        2           1000000000  0                352      0         0
rrtest          4       RUNNABLE    Normal      RR          4           1000000000  3                350      3         15
rrtest          5       ZOMBIE      Normal      RR          0           1000000000  3                338      2         14
rrtest          6       RUNNING     Normal      RR          0           1000000000  3                341      2         14
rrtest          7       RUNNABLE    Normal      RR          10          1000000000  3                344      3         12
rrtest          8       RUNNABLE    Normal      RR          7           1000000000  3                347      3         12
RR proc 6  yielded, RR ticks: 3
  busyjob 7 finished.
```

```
  busyjob 7 finished.
-----------------------
Tick: 356
-----------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  0                2        0         0
sh              2       SLEEPING    Normal      RR          0           1000000000  0                286      0         0
rrtest          3       RUNNABLE    Normal      FCFS        5           1000000000  0                352      0         0
rrtest          4       RUNNABLE    Normal      RR          7           1000000000  3                350      3         15
rrtest          5       ZOMBIE      Normal      RR          0           1000000000  3                338      2         14
rrtest          6       RUNNABLE    Normal      RR          2           1000000000  3                355      3         15
rrtest          7       RUNNING     Normal      RR          0           1000000000  3                344      2         14
rrtest          8       RUNNABLE    Normal      RR          10          1000000000  3                347      3         12
RR proc 7  yielded, RR ticks: 3
  busyjob 8 finished.
-----------------------
Tick: 360
-----------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
RR proc 8  yielded, RR ticks: 3
-----------------------------------------------------------------------------------------------------------------------------
Process_Name    PID     State       Class       Algorithm   Wait_time   Deadline    Consecutive_run  Arrival  RR_time   Running_Time
init            1       SLEEPING    Normal      RR          0           1000000000  0                2        0         0
sh              2       SLEEPING    Normal      RR          0           1000000000  0                286      0         0
rrtest          3       RUNNABLE    Normal      FCFS        9           1000000000  0                352      0         0
rrtest          4       RUNNING     Normal      RR          0           1000000000  3                350      0         15
rrtest          5       ZOMBIE      Normal      RR          0           1000000000  3                338      2         14
rrtest          6       RUNNABLE    Normal      RR          6           1000000000  3                355      3         15
rrtest          7       RUNNABLE    Normal      RR          3           1000000000  3                358      3         15
rrtest          8       RUNNABLE    Normal      RR          0           1000000000  3                361      3         15
init            1       SLEEPING    Normal      RR          0           1000000000  0                2        0         0
sh              2       SLEEPING    Normal      RR          0           1000000000  0                286      0         0
rrtest          3       RUNNABLE    Normal      FCFS        10          1000000000  0                352      0         0
rrtest          4       ZOMBIE      Normal      RR          0           1000000000  3                350      1         16
rrtest          5       ZOMBIE      Normal      RR          0           1000000000  3                338      2         14
rrtest          6       ZOMBIE      Normal      RR          0           1000000000  3                355      0         15
rrtest          7       ZOMBIE      Normal      RR          0           1000000000  3                358      0         15
rrtest          8       RUNNING     Normal      RR          0           1000000000  3                361      0         15
  busyjob 8008 finished.
```

نهایتا پردازه مادر هم اجرا شده و تست به اتمام می رسد. در تصویر بالا جزئیات زمان اجرای پردازه ها آورده شده است.