

Project3 report

810101401	رضا چهرقانی
810101540	امیر نداف فهمیده
810101575	مصطفی کرمانی نیا

مخزن گیتهاب این پروژه:

https://github.com/mostafa-kermaninia/OS_LAB_P3

آخرین کامیت:

ca37f725c16238282bbdc343daee8f4229559b80

پرسش 1) ساختار PCB و همچنین وضعیت‌های تعریف شده برای هر پردازه را در xv6 پیدا کرده و گزارش کنید. آیا شباهتی میان داده‌های موجود در این ساختار و ساختار کشیده شده در شکل ۳.۳ 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 (running)، منتظر (ready)، 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 registers: تعداد و نوع ثبات‌ها بسته به معماری کامپیوتر متفاوت است. این شامل general-purpose registers، accumulators، index registers، stack pointers هر گونه اطلاعات کد 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.

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

مدیریت حافظه (Memory-management information): این اطلاعات ممکن است شامل مواردی مانند segment table، base and limit registers، page tables باشد، بسته به سیستم حافظه‌ای که توسط سیستم‌عامل استفاده می‌شود.

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

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

```
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[NFILE];             // Open files
    struct inode *cwd;                     // Current directory
    char name[16];                          // Process name (debugging)
};
```

این ساختار، Process Control Block در xv6 است. شامل تمام اطلاعات لازم درباره یک پردازش می باشد، مانند وضعیت آن، چیدمان حافظه، فایل های باز و غیره. که خلاصه ای از آنها را بیان می کنیم

- memory limits معادل با uint sz است
- page table پوینتر به pde_t* pgdir است و برای مدیریت حافظه مجازی استفاده می شود
- Memory-Management Information این بخش هم تا حدی معادل با بخش های مربوط به

- process کرنل پشتہ پایین به پوینتر char *kstack است
- procstate enum procstate state وضعیت فعلی پردازش (مثلاً UNUSED, RUNNABLE, و غیره)
- process state و معادل با است

- pid شناسه پردازش (Process ID) است که پردازش را به طور یکتا شناسایی می کند
- process number معادل با است
- parent struct proc *parent پوینتر به پردازش والد است.

ثباتها استفاده می شود و شامل program counter هم می شود و می توان آن را ازین لحاظ معادل با program counter دانست و از طرف دیگر چون شامل اطلاعات تمام رجیسترها هم می شود، میتوان آن را معادل با cpu registers دانست

آن context struct مورد استفاده توسط swtch() برای ذخیره و بازیابی آن registers پردازش است. و در اصل رجیسترها با مقادیرشان در آن هستند پس و معادل با registers است.

اگر غیر صفر باشد، پردازش روی این کanal sleeping است void *chan ●
اگر غیر صفر باشد، نشان می دهد که پردازش به حالت killed int killed ●
آرایه‌ای از پوینترها به فایل‌های باز است و معادل با struct file *ofile[NOFILE] ●
open files ●

پوینتری به دایرکتوری فعلی است struct inode *cwd ●
نام پردازش است و غالبا برای دیباگ کردن به کار می رود. char name[16] ●

پرسش 2) هر کدام از وضعیت‌های تعریف شده معادل کدام وضعیت در شکل ۱ می‌باشند؟

استیت های موجود در xv6:

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

USED: در این وضعیت، پردازش هنوز در سیستم جای نگرفته و در حال استفاده نیست. در واقع، هنوز وارد چرخه اصلی کارکرد سیستم عامل نشده و هیچ نقشی در اجرای برنامه‌ها ایفا نمی‌کند. نهاييا بطور حدودي می توان آن را معادل با new گذاشت چون اين وضعیت خانه هاي از جدول پردازه ها است که پردازه اي در آن ها قرار ندارد.

PCB : پردازه تازه ساخته شده که هنوز آماده اجرا نیست و مقادیر PCB والد در EMBRYO -> new آن کپی نشده است.

SLEEPING -> waiting : اگر یک پردازه منتظر رخداد خاصی باشد، مانند تکمیل یک عملیات I/O یا دریافت یک سیگнал مشخص، در حالت SLEEPING به سر می برد. در این وضعیت، پردازه فعالانه کاری انجام نمی دهد و منتظر است شرایط لازم برای ادامه فعالیتش فراهم شود.

RUNNABLE -> ready : پردازه هایی که آماده اجرای روی CPU هستند اما هنوز در صفحه زمانبندی قرار دارند، به عنوان RUNNABLE شناخته می شوند. این پردازشها تمامی پیش نیازهای اجرا را دارند و تنها منتظر اختصاص CPU برای شروع فعالیت هستند.

RUNNING -> running : وقتی نوبت یک پردازه برای اجرا روی CPU فرا می رسد و عملیات محاسباتی یا دستورالعمل های برنامه را فعالانه پیش می برد، در حالت RUNNING قرار دارد.

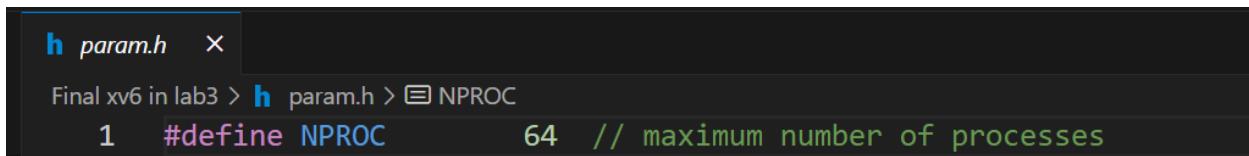
ZOMBIE -> terminated : پردازه ای که اجرایش تمام شده ولی هنوز در جدول پردازه ها وجود دارد چون والد مقدار برگشتی آن را دریافت نکرده است که آن فرزنده را clean up کند.

پرسش 3) با توجه به توضیحات گفته شده، کدام یک از توابع موجود در proc.c منجر به انجام گذار از حالت new به حالت ready که در شکل ۱ به تصویر کشیده شده، خواهد شد؟ وضعیت یک پردازه در XV6 در این گذار از چه حالت / حالت هایی به چه حالت / حالت هایی تغییر می کند؟ پاسخ خود را با پاسخ سوال ۲ مقایسه کنید.

تابع fork در حالت کلی و تابع userinit برای اولین پردازه (initproc) در ابتدا به کمک تابع allocproc یه پردازه جدید از جدول پردازه ها ptable که دارای حالت UNUSED است (یعنی آماده تولید شدن است)، اختصاص می دهند و حالت آن EMBRYO معادل new خواهد شد. شمارنده پردازه ها را یک افزایش می دهد و برخی مقداردهی ها را برای پردازه صورت می گیرد و استک آن تولید می شود. در انتها حالت آن را به RUNNABLE تغییر می دهند. البته که در init، userinit، پردازه را مقداردهی اولیه کرده اما در fork مقادیر پردازه parent را در PCB پردازه‌ی جدید می ریزد.

پرسش 4) سقف تعداد پردازه‌های ممکن در XV6 چه عددی است؟ در صورتی که یک پردازه تعداد زیادی پردازه فرزند ایجاد کند و از این سقف عبور کند، کرنل چه واکنشی نشان داده و برنامه سطح کاربر چه بازخوردی دریافت می‌کند؟

سقف تعداد پردازه‌ها NPROC برابر با 64 می‌باشد بطور دیفالت:



```
h param.h x
Final xv6 in lab3 > h param.h > NPROC
1 #define NPROC           64 // maximum number of processes
```

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

پرسش 5) چرا نیاز است در ابتدای هر حلقه تابع scheduler، جدول پردازها 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 به طور 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 وقفه‌ها کرنل می‌توانند تغییراتی در وضعیت پردازه‌ها ایجاد کنند. به عنوان مثال، حین اجرای تابع scheduler ممکن است وقفه‌ای رخ دهد و با اجرای ISR، تغییری در وضعیت پردازه‌ها صورت بگیرد پس این حالت هم منجر به ناهمگامی داده‌ها می‌شود.

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

```

328 void
329 scheduler(void)
330 {
331     struct proc *p;
332     struct cpu *c = mycpu();
333     c->proc = 0;
334
335     for(;;){
336         // Enable interrupts on this processor.
337         sti();
338
339         // Loop over process table looking for process to run.
340         acquire(&ptable.lock);
341         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
342             if(p->state != RUNNABLE)
343                 continue;
344
345             // Switch to chosen process. It is the process's job
346             // to release ptable.lock and then reacquire it
347             // before jumping back to us.
348             c->proc = p;
349             switchuvm(p);
350             p->state = RUNNING;
351
352             swtch(&(c->scheduler), p->context);
353             switchkvm();
354
355             // Process is done running for now.
356             // It should have changed its p->state before coming back.
357             c->proc = 0;
358         }
359         release(&ptable.lock);
360     }
361 }
362 }
363

```

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 نرسیده باشیم، که اگر این 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 در حالت دوم اگر در حلقه `for` داخلی، از این پردازه عبور کرده باشیم چون قبل از حالت `RUNNABLE` نبوده آن را اجرا نکردیم و در ادامه این `iteration` دوباره حالت این پردازه را چک نمی کنیم اما در `iteration` بعدی وقتی به این پردازه `iteration` بررسیم، چون در حالت `RUNNABLE` هست آن را اجرا می کنیم. پس در این حالت در `iteration` بعدی این پردازه امکان `schedule` پیدا می کند.

پرسش 7) رجیسترهاي موجود در ساختار context را نام بيريد.

```
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 پردازه‌ها استفاده می‌شود.

(EBP)Extended Base Pointer: رجیستر اشاره‌گر پایه که برای مدیریت فریم‌های استک و ارجاع به داده‌ها در استک (مثلًا 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.

پرسش 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؟

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

در سیستم‌عامل‌های مانند XV6، زمانی که تابع swtch برای تعویض متن بین پردازش‌ها در scheduler یا در حین فرآخوانی yield اجرا می‌شود، مقدار EIP در استک ذخیره می‌شود. این ذخیره‌سازی مانند فرآخوانی توابع معمولی است که در آن رجیسترها، به ویژه EIP، در استک نگهداری می‌شوند تا وضعیت پردازش جاری حفظ شود.

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

این فرایند ذخیره و بازیابی EIP به همراه دیگر رجیسترها در کد اسمبلی موجود در swtch.S مدیریت می‌شود.

پرسش 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 وقفه‌ها فعال نمی‌شد چه مشکلی به وجود می‌آمد؟

در سیستم‌عامل‌های مانند XV6، تابع sti() در ابتدای هر دور از حلقه‌ی scheduler برای فعال‌سازی وقفه‌ها فرآخوانی می‌شود. اگر این وقفه‌ها فعال نشوند، قابلیت preemption timer که به به وابسته است از بین می‌رود. این بدین معناست که هیچ‌گونه تعویض متن (context switch) نمی‌تواند انجام شود، حتی اگر پردازش‌هایی آماده برای اجرا (RUNNABLE) وجود داشته باشند.

یک 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 یا سایر وقفه‌ها خواهند ماند.

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

پرسش 10) بنظر شما وقفه تایمر هر چه مدت یک بار صادر می‌شود؟ راهنمایی می‌توانید با اضافه کردن یک `cprintf` پس از `+ + ticks` این موضوع را مشاهده کنید.

بر اساس داکیومنت xv6، در هر ثانیه ۱۰۰ تایمر یک interrupt تولید می‌کند.

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 10ms یک بار صادر می‌شود.

این نتیجه را می‌توان طبق راهنمایی سوال و با افزودن `cprintf` بعد از هر بار `++tick` هم گرفت، کافیست مثلاً با تابع `cmosetime`، زمان آنی را در دو `tick` متواتی بدست آورده و فاصله‌ی آن‌ها از هم را بیابیم که البته چون به واحد ثانیه زمان را به ما می‌دهد، می‌توان صبر کرد تا فاصله‌ی دو زمان

سنجی، 1 ثانیه شود و سپس تعداد tick ها را در یک ثانیه داریم و با میانگین گرفتن از آن، دقیقا به همین مقدار 100 بار در ثانیه می‌رسیم.

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

```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ticks: 42
ticks: 140
ticks: 240
ticks: 340
ticks: 440
ticks: 540
ticks: 640
```

همانطور که دیده می‌شود به طور میانگین در هر ثانیه صد تیک اضافه می‌شود که این به این معنی است که `++ticks` هر 10ms یک بار اجرا می‌شود.

پرسش 11) با توجه به توضیحات داده شده، چه تابعی منجر به انجام شدن گذار 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.

پرسش 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، کوانتم 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 صدا زده می‌شود و پردازنه از پردازه فعلی گرفته می‌شود و به پردازه 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();
```

چونکه وقفه تایمر هر 10ms یک بار صادر می‌شود پس کوانتم زمانی برابر با 10ms ثانیه خواهد شد.

پرسش 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 بازگرداند.

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 `wait` از تابع `sleep` استفاده می‌کند. تابع `sleep` پردازه والد را به حالت خواب (sleep) در می‌آورد 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:

- اشاره‌گر به پردازه والد است که منتظر می‌ماند.
- قفل جدول پردازه‌ها است که باعث می‌شود پردازه والد به هنگام انتظار 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.

پرسش 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 تایмер/تاخیر: پردازه‌ها می‌توانند برای مدت مشخصی به خواب بروند، مانند استفاده از `sys_sleep` برای ایجاد تاخیر زمانی.

```
int 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);
```

همگام‌سازی در درایورهای دستگاه: پردازه‌ها در حین انتظار برای عملیات O/I ، 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);
```

- همگام سازی پردازش‌ها با رویدادها: پردازه‌ها هنگام انتظار برای یکسری رویداد‌ها مثل نوشته شدن یا خوانده شدن داده از منابع مانند لوله‌ها (pipe)، به خواب می‌روند.

```
int
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
        }
    }
}
```

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

در سیستم‌عامل XV6، تابع wakeup برای بیدار کردن پردازه‌های استفاده می‌شود که منتظر یک رویداد خاص بوده‌اند. این تابع ابتدا قفل جدول پردازه‌ها (ptable.lock) را می‌گیرد و سپس تابع wakeup1 را صدای زند تا پردازه‌های منتظر را بیدار کند.

```

// 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 پردازه ها را پیمایش کرده و پردازه ها را از حالت `SLEEPING` تغییر وضعیت می دهد، تنها در صورتی که پردازه منتظر همان رویدادی باشد که در `wakeup1` مشخص شده است. این بررسی با مقایسه متغیر `chan` پردازه با داده ای که به `chan` ارسال شده انجام می شود.

هر پردازه ای می تواند با استفاده از `wakeup` پردازه دیگر را بیدار کند و در کل تابع `wakeup` باعث می شود پردازه هایی که منتظر رویدادی خاص بودند، پس از وقوع آن رویداد، به وضعیت `RUNNABLE` تغییر کنند و آماده اجرا شوند.

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

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

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

در سیستم عامل XV6، علاوه بر تابع wakeup (که پردازه را از وضعیت SLEEPING تغییر می دهد، تابع kill) نیز می تواند این گذار را انجام دهد. وقتی پردازه ای در وضعیت RUNNABLE باشد و سیگنال خاتمه دریافت کند، تابع kill (وضعیت آن را به خواب SLEEPING) تغییر می دهد تا پردازه قادر به پاسخ دادن به سیگنال خاتمه باشد.

کد تابع kill:

```

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.

پرسش 18) در بخش ۳.۳.۲ منبع درس با پردازه‌های Orphan آشنا شدید، رویکرد xv6 در رابطه با این گونه پردازه‌ها چیست؟

پردازه هایی که اجرای والد آن ها به هر دلیلی زودتر از پردازه فرزند تمام می شود، پردازه یتیم یا Orphan نامیده می شوند. در xv6، پردازه های فرزند زمانی که تمام می شوند به حالت ZOMBIE می روند تا توسط والد خود اطلاعاتشان جمع آوری شوند. اما اگر یک پردازه یتیم شده باشد، والدی وجود ندارد که اطلاعات این پردازه را جمع آوری کند. در xv6 زمانی که یک پردازه به حالت ZOMBIE می رسد ولی والد آن موجود نیست، به پردازه initproc که همواره در حال اجرا است ارجاع داده می شود و اطلاعات آن توسط پردازه initproc جمع آوری می شود و این فرزند ها از initproc پاک می شوند. این روند باعث می شود که پردازه ptable ، در فرآخوانیتابع wait توسط initproc اشغال نکنند.

```

291
292     // Pass abandoned children to init.
293     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
294     {
295         if (p->parent == curproc)
296         {
297             p->parent = initproc;
298             if (p->state == ZOMBIE)
299                 wakeup1(initproc);
300         }
301     }
302 }
```

پرسش 19) مقدار CPUS را مجددا به عدد ۲ برگردانید. آیا همچنان ترتیبی که قبل مشاهده میکردید پا بر جاست؟ علت این امر چیست؟

برنامه سطح کاربری می‌نویسیم که درون خود سه پردازه ایجاد می‌کند.

CPUS = 1

```
cpu:0 tick:738 pid:6 queue:0 arrival:485
cpu:0 tick:739 pid:4 queue:0 arrival:483
cpu:0 tick:740 pid:4 queue:0 arrival:483
cpu:0 tick:741 pid:4 queue:0 arrival:483
cpu:0 tick:742 pid:4 queue:0 arrival:483
cpu:0 tick:743 pid:4 queue:0 arrival:483
cpu:0 tick:744 pid:5 queue:0 arrival:484
cpu:0 tick:745 pid:5 queue:0 arrival:484
cpu:0 tick:746 pid:5 queue:0 arrival:484
cpu:0 tick:747 pid:5 queue:0 arrival:484
cpu:0 tick:748 pid:5 queue:0 arrival:484
cpu:0 tick:749 pid:6 queue:0 arrival:485
cpu:0 tick:750 pid:6 queue:0 arrival:485
cpu:0 tick:751 pid:6 queue:0 arrival:485
cpu:0 tick:752 pid:6 queue:0 arrival:485
cpu:0 tick:753 pid:6 queue:0 arrival:485
cpu:0 tick:754 pid:4 queue:0 arrival:483
```

CPUS = 2

```
cpu:0 tick:738 pid:5 queue:0 arrival:521
cpu:1 tick:738 pid:4 queue:0 arrival:520
cpu:0 tick:739 pid:5 queue:0 arrival:521
cpu:1 tick:739 pid:4 queue:0 arrival:520
cpu:0 tick:740 pid:5 queue:0 arrival:521
cpu:1 tick:740 pid:4 queue:0 arrival:520
cpu:0 tick:741 pid:5 queue:0 arrival:521
cpu:1 tick:741 pid:4 queue:0 arrival:520
cpu:0 tick:742 pid:5 queue:0 arrival:521
cpu:1 tick:742 pid:4 queue:0 arrival:520
cpu:0 tick:743 pid:6 queue:0 arrival:524
cpu:1 tick:743 pid:5 queue:0 arrival:521
cpu:0 tick:744 pid:6 queue:0 arrival:524
cpu:1 tick:744 pid:5 queue:0 arrival:521
cpu:0 tick:745 pid:6 queue:0 arrival:524
cpu:1 tick:745 pid:5 queue:0 arrival:521
cpu:0 tick:746 pid:6 queue:0 arrival:524
```

هنگامی که تعداد پردازنده ها برابر یک است، در هر نوبت گردشی پردازنده از پردازه فعلی گرفته شده و به پردازه بعدی اختصاص داده می‌شود و از آنجایی که هر کوانتوم زمانی برابر با ۵ تیک زمانی می‌باشد، بنابراین پس از پنج تیک زمانی به پردازه بعد می‌رویم و همینطور به ترتیب پردازه ها را اجرا می‌کنیم.

اما زمانی که دو پردازنده داریم، هر پردازنده به یک پردازه اختصاص پیدا می‌کند و از آنجایی که پردازنده ها به صورت موازی کار می‌کنند پس دو پردازه اختصاص یافته همزمان اجرا می‌شوند و در هر تیک زمانی شاهد اجرای دو پردازنده روی دو پردازنده متفاوت هستیم. در این حالت چون هر پردازنده scheduler خودش را دارد، پس از اتمام کوانتوم زمانی به پردازه‌ی بعدی قابل اجرا و این دو به صورت مجزا از هم یک برنامه را انتخاب و اجرا می‌کنند.

پرسش 20) در صورت نیاز به مقداردهی اولیه به فیلدهای اضافه شده در ساختار `cpu`، در چه تابعی از `xv6` بهتر است این کار انجام گیرد؟ راهنمایی به `main.c` مراجعه کنید)

مقدار دهی اولیه به مقادیر پردازنده بهتر از قبل از شروع به کار آن انجام شود. پس در تابع mpmain قبل از آنکه پردازنده شروع به کار کند (هنگامی که فیلد started را یک می‌کنیم) به فیلدها مقدار اولیه می‌دهیم. مزیت دیگر تابع mpmain آن است که برای همه پردازنده‌ها یکسان می‌باشد. یعنی تمامی پردازنده‌ها (هم پردازنده اولیه و هم سایر پردازنده‌ها که اضافه می‌شوند) در این تابع مشترک هستند.

```
// Common CPU setup code.  
static void  
mpmain(void)  
{  
    mycpu()->schedqueue = RR;  
    mycpu()->queueticks = 0;  
    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  
}
```

پرسش 21) با توجه به سیاست‌های پیاده‌سازی شده در سطح‌های دوم و سوم و همچنین استفاده از روش time-slicing توجیه کنید چرا همچنان مشکل starvation امکان رخ دادن دارد؟

در صفحه دوم اگر پردازه‌هایی با Confidence کم و Burst Time زیاد (مثلاً ۰.۰۱) وجود داشته باشد، دیگر هرگز نوبت به پردازه‌ای که Burst Time زیادی دارد نمی‌رسد زیرا همواره پردازه‌ای با Burst Time کمتر وجود دارد که با احتمال بالا (۰.۰۱ درصد) انتخاب می‌شود در نتیجه starvation می‌دهد.

در صفحه سوم هم اگر پردازه‌ای مدت زمان زیادی اجرا شود (مثلاً حلقه while بی‌نهایت داشته باشد) هرگز نوبت به پردازه‌هایی که بعد از آن وارد این صفحه شده‌اند نمی‌رسد.

همچنین time-slicing هیچ تاثیری بر روی starvation ندارد. برای مثال در صفحه دوم پس از آنکه پردازنده دوباره به این صفت اختصاص داده شد، پردازه ها با Burst Time کمتر انتخاب می شوند و باز هم نوبت به پردازه با Burst Time زیاد نمی رسد. در صفحه سوم هم باز هم همان پردازه های که در time slice قبلی داشت اجرا می شد، به اجرایش ادامه می دهد.

پرسش 22) به چه علت مدت زمانی که پردازه در وضعیت SLEEPING میباشد به عنوان زمان انتظار پردازه از منظر زمان بندی 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 یک 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 پردازه های CPU-bound جلوگیری کنیم. اگر پردازه های SLEEPING را در افزایش زمان انتظار دخیل کنیم، صفت آنها با اینکه در انتظار cpu نیستند تغییر می کند و این باعث می شود پردازه های CPU-bound مدت زیادی منتظر cpu بمانند و همچنان فرصت اجرای کمتری داشته باشند (چون اولویت پردازه های SLEEPING هم افزایش پیدا کرده است).

زمان بندی بازخورد چند سطحی:

سطح اول: زمانبند نوبت گردشی با کوانتم زمانی

برای افزایش نوبت گردشی به ۰.۵ میلی ثانیه، تابع yield را هنگامی که باقیمانده تعداد ticks صفر نوبت گردشی بر ۵ صفر شود صدای زنیم.

```
} else if(mycpu()->schedqueue == RR && mycpu()->queueticks % 5 == 0)
    yield();
```

برای پیاده سازی الگوریتم، از پردازه بعدی که نگه داشته ایم شروع می کنیم و در صورت RUNNABLE بودن و عضو بودن در این صفحه، پردازه ای را که به عنوان پردازه بعدی می خواهیم اجرا کنیم برابر آن قرار می دهیم. این کار تا زمانی که یک دور کامل روی ptable بزنیم، انجام می دهیم.

```
case RR:
    // Loop over process table looking for process to run.
    p = nexrrp;
    do
    {
        if (p->state != RUNNABLE || p->schedqueue != RR)
        {
            if (++p == &ptable.proc[NPROC])
                p = ptable.proc;
            continue;
        }

        nextp = p;

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

سطح دوم: اول کوتاهترین کار

بر روی تمامی پردازه های RUNNABLE و عضو این صفت پیمایش می کنیم، در صورتی که burst time آن از پردازه انتخاب شده کمتر باشد، با تولید یک عدد رندوم و چک کردن آن با confidence اگر مقدار تولید شده کمتر بود، پردازه جدید را به عنوان پردازه انتخابی یعنی nextp قرار می دهیم. در حالی که هیچ پردازه ای بر اساس confidence آن انتخاب نشود، طبق قرارداد باید بلندترین کار را انتخاب کنیم. پس برای این کار در همین پیمایش longestjob را هم مقدار دهی می کنیم که در صورت انتخاب نشدن هیچ یک از پردازه ها، آن را انتخاب کنیم.

```
case SJF:
    longestjob = 0;
    // Loop over process table looking for the shortest process to run.
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state != RUNNABLE || p->schedqueue != SJF)
            continue;

        if (nextp == 0 || p->bursttime < nextp->bursttime)
            if (rand() % 100 < p->confidence)
                nextp = p;

        if (longestjob == 0 || p->bursttime > longestjob->bursttime)
            longestjob = p;
    }
    if (nextp == 0)
        nextp = longestjob;
    break;
```

سطح سوم: اولین ورود-اولین رسیدگی

در ساختار داده پردازه struct proc یک خانه fcfseentry قرار می‌دهیم تا هنگامی که آن را وارد صفت FCFS می‌کنیم با این مقدار مشخص کنیم که چندمین پردازه‌ای است که وارد این صفت شده است. در واقع به کمک مقدار این فیلد متوجه می‌شویم که کدام پردازه زودتر وارد صفت شده است. برای مقدار دهی به آن، هر زمان که پردازه‌ای را وارد این صفت می‌کنیم به صورت زیر مقدار می‌دهیم:

```
p->schedqueue = FCFS;
p->fcfseentry = nextfcfs++;
```

برای انتخاب یک پردازه از این صفت، یک بار کل پردازه‌ها را پیمایش کرده و آن را که کوچکترین را دارد به عنوان nextp می‌دانیم.

```
case FCFS:
    // Loop over process table looking for the first process to run.
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state != RUNNABLE || p->schedqueue != FCFS)
            continue;

        if (nextp == 0 || p->fcfseentry < nextp->fcfseentry)
            nextp = p;
    }
    break;
}
```

اگر دیگر پردازه‌ای در صفت برای انتخاب شدن وجود نداشته باشد یعنی اینکه nextp مقدار دهی نشده است، در این صورت صفت پردازنده را به صفت بعدی تغییر می‌دهیم و زمان را صفر می‌کنیم تا زمان سپری شده در این صفت در زمان صفت بعدی تأثیر نگذارد.

```

// start next queue if queue is empty
if (nextp == 0)
{
    c->schedqueue = (c->schedqueue + 1) % NSCHEDQUEUE;
    c->queueticks = 0;
}
else
    switch_to_chosen_process(nextp, c);

```

برشدهی زمانی

به ساختار داده پردازنده struct cpu دو فیلد schedqueue و queueticks را قرار می‌دهیم تا مشخص کنند که پردازنده در کدام صف و چند تیک زمانی در این صف قرار دارد. برای جابه‌جا شدن پردازنده بین صفحات، در تابع trap هنگامی که وقفه تایمر صادر می‌شود، چک می‌کنیم که آیا در صف فعلی زمانش تمام شده است یا خیر. اگر تمام شده بود که باید به صف بعدی برود پس با تابع yield پردازنده را از پردازه فعلی می‌گیریم. اگر هم زمانش تمام نشده بود ولی در صف نوبت‌هی گردشی قرار داشت، چک می‌کنیم که اگر کوانتم زمانی تمام شده باشد پردازنده را به پردازه‌ی بعدی در این صف بدهیم.

```

if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER){
    mycpu()->queueticks++;
    if((mycpu())->schedqueue == RR && mycpu()->queueticks == 30) ||
       (mycpu()->schedqueue == SJF && mycpu()->queueticks == 20) ||
       (mycpu()->schedqueue == FCFS && mycpu()->queueticks == 10)){
        mycpu()->schedqueue = (mycpu()->schedqueue + 1) % NSCHEDQUEUE;
        mycpu()->queueticks = 0;
        yield();
    } else if(mycpu()->schedqueue == RR && mycpu()->queueticks % 5 == 0)
        yield();
}

```

سازوکار افزایش سن

برای این بخش نیاز هست که هر زمان `++tick` اتفاق می‌افتد یکی به `wait_time` هر پردازه‌ای که در حالت RUNNABLE هست اضافه کنیم سپس چک کنیم که اگر به 800 رسید صف آن را عوض می‌کنیم. برای این کار تابع `age_processes` را در `proc.c` به صورت زیر می‌نویسیم دیکلریشن این تابع در فایل `def.h` قرار دارد. دقت شود که `proc` در استراکت `wait_time` قرار دارد و در ابتدا به صفر اینیشیالایز می‌شود. همچنین در زمان عوض کردن صف، `wait_time` دوباره به صفر برمی‌گردد و همچنین `arrival_time` هم آپدیت می‌شود.

```
void age_processes(void)
{
    acquire(&ptable.lock);
    struct proc *p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state != RUNNABLE)
            continue;
        p->wait_time++;
        if (p->wait_time == 800)
        {
            switch (p->schedqueue)
            {
                case FCFS:
                    p->wait_time = 0;
                    p->schedqueue = SJF;
                    p->arrival = ticks;
                    cprintf("pid:%d perv_queue:FCFS new_queue:SJF\n", p->pid);
                    break;
                case SJF:
                    p->wait_time = 0;
                    p->schedqueue = RR;
                    p->arrival = ticks;
                    cprintf("pid:%d perv_queue:SJF new_queue:RR\n", p->pid);
                    break;
                default:
                    break;
            }
        }
    }
    release(&ptable.lock);
}
```

سپس در trap.c بعد از ++ticks این تابع را فراخوانی می کنیم.

```
switch(tf->trapno){  
case T_IRQ0 + IRQ_TIMER:  
    if(cpuid() == 0){  
        acquire(&tickslock);  
        ticks++;  
        age_processes();
```

فراخوانی های سیستمی مورد نیاز

در ابتدا در فایل های syscall.h و syscall.S این سه فراخوانی را اضافه می کنیم.

36 SYSCALL(list_all_processes), 37 SYSCALL(change_queue) 38 SYSCALL(processes_info) 39 SYSCALL(set_bc) 40	27 #define SYS_list_all_processes 26 28 #define SYS_change_queue 27 29 #define SYS_processes_info 28 30 #define SYS_set_bc 29 31
--	--

سپس تابع های مربوطه رو در c دیگر میکنیم.

103 extern int sys_wait(void); 104 extern int sys_write(void); 105 extern int sys_uptime(void); 106 extern int sys_create_palindrome(void); 107 extern int sys_move_file(void); 108 extern int sys_sort_syscalls(void); 109 extern int sys_list_all_processes(void); 110 extern int sys_get_most_invoked_syscall(void); 111 extern int sys_change_queue(void); 112 extern int sys_processes_info(void); 113 extern int sys_set_bc(void); 114	125 [SYS_dup] sys_dup, 126 [SYS_getpid] sys_getpid, 127 [SYS_sbrk] sys_sbrk, 128 [SYS_sleep] sys_sleep, 129 [SYS_uptime] sys_uptime, 130 [SYS_open] sys_open, 131 [SYS_write] sys_write, 132 [SYS_mknod] sys_mknod, 133 [SYS_unlink] sys_unlink, 134 [SYS_link] sys_link, 135 [SYS_mkdir] sys_mkdir, 136 [SYS_close] sys_close, 137 [SYS_create_palindrome] sys_create_palindrome, 138 [SYS_move_file] sys_move_file, 139 [SYS_sort_syscalls] sys_sort_syscalls, 140 [SYS_get_most_invoked_syscall] sys_get_most_invoked_syscall, 141 [SYS_list_all_processes] sys_list_all_processes, 142 [SYS_change_queue] sys_change_queue, 143 [SYS_processes_info] sys_processes_info, 144 [SYS_set_bc] sys_set_bc, 145 };
---	---

سپس در sysproc.c این توابع را به این صورت تعریف می کنیم که آرگومان های لازم را بگیرند و سپس آن ها را به توابعی که در proc.c نوشتهیم پاس دهند.

```
15 int sys_change_queue(void){
16     int pid, sel_q;
17     if (argint(0, &pid) < 0)
18         return -1;
19     if (argint(1, &sel_q) < 0)
20         return -1;
21     change_queue(pid, sel_q);
22     return 0;
23 }
24
25 int sys_processes_info(void){
26     processes_info();
27     return 0;
28 }
29
30 int sys_set_bc(void){
31     int pid, bursttime, confidence;
32     if (argint(0, &pid) < 0)
33         return -1;
34     if (argint(1, &bursttime) < 0)
35         return -1;
36     if (argint(2, &confidence) < 0)
37         return -1;
38     set_bc(pid, bursttime, confidence);
39     return 0;
40 }
```

حال دونه دونه توابع را در proc.c تعریف می کنیم.

:change_queue

در این تابع ابتدا چک می کنیم که شماره صف به درستی وارد شده باشد. اگر درست بود به دنبال پروسس با pid مورد نظر می گردیم. وقتی این پروسس را پیدا کردیم اگر صف انتخاب شده با صف کنونی پروسس یکی نبود، ابتدا یک لاغ پرینت می کنیم که شامل pid، شماره صف قبلی، شماره صف جدید است، سپس arrival و صف این پروسس را آپدیت می کنیم.

```
804 void change_queue(int pid, int chosen_q)
805 {
806     if (chosen_q < 0 || chosen_q > 2)
807     {
808         cprintf("Invalid queue\n");
809         return;
810     }
811
812     struct proc *p;
813     acquire(&pstable.lock);
814     for (p = ptable.proc; p < &pstable.proc[NPROC]; p++)
815     {
816         if (p->pid == pid && chosen_q != p->schedqueue)
817         {
818             cprintf("pid: %d perv_q:%d new_q:%d\n", pid, p->schedqueue, chosen_q);
819             p->arrival = ticks;
820             p->schedqueue = chosen_q;
821         }
822     }
823     release(&pstable.lock);
824 }
825
826 }
```

:processes_info

در این تابع به ازای تمام پردازه های موجود در ptable به غیر از item هایی که در استیت UNUSED هستند (یعنی در واقع پروسسی در این خانه ذخیره نشده) ابتدا state پردازه را به صورت string در می آوریم سپس اطلاعات خواسته شده را پرینت می کنیم.

```
void processes_info(void)
{
    struct proc *p;

    acquire(&ptable.lock);
    cprintf(".....\n");
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state != UNUSED)
        {
            char *state_name;
            switch (p->state)
            {
                case EMBRYO:
                    state_name = "EMBRYO";
                    break;
                case SLEEPING:
                    state_name = "SLEEPING";
                    break;
                case RUNNABLE:
                    state_name = "RUNNABLE";
                    break;
                case RUNNING:
                    state_name = "RUNNING";
                    break;
                case ZOMBIE:
                    state_name = "ZOMBIE";
                    break;
                default:
                    state_name = "UNKNOWN";
                    break;
            }
        }
    }
}
```

```
    cprintf("name:%s pid:%d state:%s queue:%d wait:%d confidence:%d burst time:%d consecutive:%d arrival:%d\n"
           , p->name, p->pid, state_name, p->schedqueue,
           p->wait_time, p->confidence, p->bursttime, p->consecutive_time, p->arrival);
}
cprintf(".....\n");
release(&ptable.lock);
}
```

دقت شود که برای run consecutive مقدار initialize برابر صفر است اما بعد از هر یک ایتریشن در تابع add_consecutive انجام می دهیم و به پردازه ای که در حالت

RUNNING باشد، یک به آن اضافه می‌کنیم. به پردازه‌ها در حالت‌های دیگر هم consecutive_time برابر صفر قرار می‌دهیم (چون در حال اجرا نیستن که بگیم چند تیک پشت هم اجرا شدن).

```
8 switch(tf->trapno){  
9     case T_IRQ0 + IRQ_TIMER:  
10         if(cpuid() == 0){  
11             acquire(&tickslock);  
12             ticks++;  
13             age_processes();  
14             add_consecutive();  
15             wakeup(&ticks);  
16         }  
17     }  
18 }  
  
void add_consecutive(){  
    acquire(&ptable.lock);  
    struct proc *p;  
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)  
    {  
        if (p->state == RUNNING)  
            p->consecutive_time++;  
        else  
            p->consecutive_time = 0;  
    }  
    release(&ptable.lock);  
}
```

:set_bc

در این تابع ابتدا پروسس با pid مورد نظر را پیدا می‌کند سپس bursttime و confidence آن را به مقادیر جدید تغییر می‌دهد.

```
809 void set_bc(int pid, int bursttime, int confidence)  
810 {  
811     struct proc *p;  
812     acquire(&ptable.lock);  
813     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)  
814     {  
815         if (p->pid == pid)  
816         {  
817             p->bursttime = bursttime;  
818             p->confidence = confidence;  
819         }  
820     }  
821     release(&ptable.lock);  
822 }  
823 }
```

سر آخر هم در user.h این سیستم کال ها را اضافه می کنیم تا با استفاده از برنامه سطح کاربر از این سیستم کال ها بتوانیم استفاده کنیم.

```
3 // system calls
4 int fork(void);
5 int exit(void) __attribute__((noreturn));
6 int wait(void);
7 int pipe(int *);
8 int write(int, const void *, int);
9 int read(int, void *, int);
10 int close(int);
11 int kill(int);
12 int exec(char *, char **);
13 int open(const char *, int);
14 int mknod(const char *, short, short);
15 int unlink(const char *);
16 int fstat(int fd, struct stat *);
17 int link(const char *, const char *);
18 int mkdir(const char *);
19 int chdir(const char *);
20 int dup(int);
21 int getpid(void);
22 char *sbrk(int);
23 int sleep(int);
24 int uptime(void);
25 void create_palindrome(void);
26 int move_file(const char *, const char *);
27 int sort_syscalls(int);
28 int list_all_processes(void);
29 int get_most_invoked_syscall(int);
30 int change_queue(int, int);
31 int processes_info(void);
32 int set_bc(int, int, int);
33
```

برنامه سطح کاربر scheduletest هم به صورت زیر نوشته شده که از تمامی این سیستم کال ها استفاده می‌کند. این برنامه ده تا پردازه با استفاده از fork ایجاد می‌کند و برای هرکدام به صورت رندوم یا صف تغییر می‌دهد یا set_bc انجام می‌کند و در زمان های معینی هم اطلاعات را چاپ می‌کند.

```
4  #define NPROC 10
5
6
7  int
8  fibonacci(int n){
9    if (n <= 1) {
10      return n;
11    }
12    return fibonacci(n - 1) + fibonacci(n - 2);
13  }
14
15  int
16  main(void)
17  [
18    int pid, n = 39;
19
20    for(int i = 0; i < NPROC; i++){
21      pid = fork();
22      if(pid < 0){
23        printf(1, "scheduletest: fork failed\n");
24        exit();
25      }
26      if(pid == 0){
27        printf(1, "fibonacci(%d) = %d\n", n, fibonacci(n));
28        exit();
29      }
30      if(pid % 3 == 0)
31        set_bc(pid, 6, 80);
32      if (pid%4 == 0)
33        change_queue(pid, 1);
34      printf(1, "scheduletest: starting process %d\n", pid);
35      processes_info();
36    }
37
38    for(int i = 0; i < NPROC; i++){
39      printf(1, "scheduletest: ending process %d\n", wait());
40      processes_info();
41    }
42
43    exit();
44 ]
```