Repository Link: <https://github.com/SamanEN/Operating-System-Lab-Projects>

Latest Commit Hash: 9f83998e108bd9f32dc7287e049282666d7b6622

# زمان‌بندی در xv6

## چرا فراخوانی sched() منجر به فراخوانی scheduler() می‌شود؟

هر هسته‌ای که شروع به کار می‌کند، تابع mpmain را صدا می‌زند. این تابع نیز در انتها تابع scheduler را صدا می‌زند که باعث شروع به کار زمان‌بند مربوط به هر هسته می‌شود. تابع ذکر شده در ptable به دنبال پردازه قابل اجرا می‌گردد و در صورت یافتن چنین پردازه‌ای (RUNNABLE)، پس از تغییر حافظه به حافظه پردازه توسط تابع switchuvm، با استفاده از تابع swtch که در زبان اسمبلی پیاده‌سازی شده، عملیات تعویض متن را انجام می‌دهد. این تابع، رجیسترهای context قدیمی (cpu::struct context \*scheduler) را در آدرس مربوط به همان context ذخیره می‌کند و رجیسترهای مربوط به context جدید را از آدرس مربوط به همان context بازیابی می‌کند که با این کار، program counter نیز به مقدار متناظر آن در context جدید تبدیل می‌شود و به این ترتیب، پردازه جدید شروع به اجرا می‌کند. در 3 حالت زیر، پردازه در حال اجرا تابع sched را فراخوانی می‌کند:

1. پردازه با استفاده از فراخوانی سیستمی exit، پردازنده را ترک کند.
2. پردازه با استفاده از فراخوانی سیستمی sleep، به حالت SLEEPING در آید.
3. پس از interrupt ایجاد شده توسط تایمر، پردازه مجبور به خروج از پردازنده شود که در این حالت تابع yield فراخوانی شده و در آن تابع نیز تابع sched فراخوانی می‌شود.

در نهایت در تابع sched، مجددا عملیات تعویض متن صورت می‌پذیرد و در این حالت contextای که در استراکت cpu (struct context \*scheduler) بازیابی می‌شود و context مربوط پردازه در حال اجرا ذخیره می‌شود. پس از بازیابی context مربوط به scheduler، program counter به خط 2782 اشاره می‌کند و باعث ادامه کار scheduler می‌شود. در واقع پردازه‌ای که هر هسته را آماده به کار می‌کند، هیچ وقت از تابع scheduler خارج نمی‌شود و فقط با عملیات context switching از پردازنده خارج می‌شود و با اجرای تابع sched، دوباره به ادامه کار خود می‌پردازد.

لازم به ذکر است که program counter به صورت مستقیم ذخیره نمی‌شود بلکه همان return adr تابع است که در زمان فراخوانی تابع swtch، در استک push می‌شود. این آدرس پس از دستور ret در خط 3078، از استک pop شده و در رجیستر مربوط به program counter قرار می‌گیرد.

# زمان‌بندی

## ساختار صف اجرا در زمان‌بند کاملاً منصف لینوکس

صف اجرا در لینوکس توسط یک red-black tree پیاده‌سازی می‌شود. در چپ‌ترین گره این درخت، پردازه‌ای قرار گرفته که کمترین برش زمانی در حین اجرا را داشته است. (این مقدار در vruntime اطلاعات پردازه ذخیره شده است)

## بررسی لینوکس و xv6 از منظر مشترک یا مجزا بودن صف‌های زمان‌بندی

هر پردازنده زمان‌بند خودش را دارد ولی همانطور که در شکل دیده می‌شود، در xv6 فقط از یک صف زمان‌بندی برای همه پردازنده‌ها به طور مشترک استفاده می‌شود:

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

این صف از struct procها حداکثر NPROC (64) پردازه همزمان را می‌تواند در خود نگه دارد.

برای جلوگیری از خرابی حاصل از تغییرات همزمان چند پردازنده روی این صف، از یک spinlock استفاده می‌شود. به این صورت که هنگام دسترسی به ptable.proc ابتدا باید ptable.lock را acquire کنیم و پس از انجام تغییرات در آن، آن را release می‌کنیم.

داشتن فقط یک صف برای همه پردازنده‌ها، پیاده‌سازی را کمی ساده‌تر می‌کند ولی در عوض به lock نیاز دارد که می‌تواند کمی بر روی performance نیز تأثیر بگذارد.

از آنجا که پردازه‌ای که در این صف است، هر بار در یک پردازنده اجرا می‌شود و بین آنها جهش می‌کند، با توجه به اینکه هر پردازنده high-level cache خودش را دارد، کارایی cache بسیار کمتر می‌شود.

در لینوکس، هر پردازنده صف زمان‌بندی مخصوص خودش را دارد و پردازه‌ها به صورت مجزا در آنها قرار می‌گیرند.

این طراحی نیازمند load balancing است یعنی توازن در همه صف‌های پردازنده‌ها برقرار باشد و یک پردازنده خالی و دیگری پر نماند. نیازی به این کار در صف مشترک نیست.

## چرا در اجرای حلقه ابتدا وقفه فعال می‌گردد؟ آیا در سیستم تک هسته‌ای به آن نیاز است؟

زمانی که قفل ptable فعال می‌شود، تمامی interruptها به وسیله تابع pushcli غیرفعال می‌شوند. حال ممکن است پردازنده در حالتی قرار بگیرد که تعدادی از پردازه‌های آن منتظر پایان عملیات I/O باشند و هیچ کدام از پردازه‌های دیگر نیز در حالت RUNNABLE نباشند. در این حالت هیچ پردازه دیگری اجرا نمی‌شود و اگر interruptها نیز هیچ وقت فعال نشود، پس از پایان عملیات I/O نمی‌توانیم پردازه‌های مربوطه را به حالت RUNNABLE تغییر دهیم که بتوانند اجرا شوند، در نتیجه سیستم فریز می‌شود. به همین دلیل است که در این حلقه برای مدت کوتاهی (تا پیش از قفل کردن ptable)، وقفه‌ها فعال می‌شوند تا در صورت نیاز بتوانیم حالت پردازه‌ها را تغییر دهیم.

## دو سطح مدیریت وقفه‌ها در لینوکس

مدیریت وقفه‌ها در لینوکس و بسیاری از سیستم عامل‌های امروزی، در دو سطح اول و دوم صورت می‌گیرد. به این دو سطح FLIH[[1]](#footnote-1) و SLIH[[2]](#footnote-2) گفته می‌شود. علاوه بر آن در لینوکس به FLIH، نیمه بالایی[[3]](#footnote-3) و به SLIH، نیمه پایینی[[4]](#footnote-4) گفته می‌شود. وظیفه FLIH مدیریت وقفه‌های ضروری در کم‌ترین زمان ممکن است؛ یا وقفه را به طور کامل سرویس‌دهی می‌کند و یا اطلاعات ضروری وقفه را -که فقط در زمان وقوع وقفه در دسترسی است- ذخیره کرده و یک SLIH را برای مدیریت کامل این وقفه زمان‌بندی می‌کند. در روال جواب‌دهی به وقفه‌ها در FLIH، یک تعویض متن[[5]](#footnote-5) صورت گرفته و کد مربوط به مدیریت‌کنندۀ وقفۀ صورت‌گرفته بارگزاری و اجرا می‌شود. FLIH می‌تواند باعث ایجاد لغزش[[6]](#footnote-6) (لَگ) در پردازه‌ها شود. علاوه بر آن FLIH باعث چشم‌پوشی[[7]](#footnote-7) کردن از وقفه‌ها می‌شود.

SLIH وظیفه بخش‌هایی از پردازش وقفه‌ها را بر عهده دارد که زمان‌بر می‌باشند؛ این کار مانند یک پردازه انجام می‌شود. SLIHها یا یک ریسه مخصوص در سطح کرنل برای هر هندلر دارند، یا توسط یک thread pool مدیریت می‌شوند. SLIHها در یک صف اجرا قرار گرفته و منتظر پردازنده می‌مانند. از آنجا که ممکن است زمان طولانی برای اجرای آن‌ها نیاز باشد، SLIHها نیز معمولا مانند ریسه‌ها و پردازه‌ها زمان‌بندی می‌شوند.

## گرسنگی پردازه‌ها چگونه حل شده است؟

گاها ممکن است پردازه‌ای، به دلیل داشتن اولویت کم‌تر نسبت به باقی پردازه‌های در حال اجرا، مدت نامشخصی را در صف آماده سپری کند؛ به این اتفاق گرسنگی پردازه گفته می‌شود. برای حل این موضوع راهکار aging ارائه شده است؛ Aging بدین معنی است که هر چه یک پردازه با اولویت کمتر در صف اجرا باقی بماند، اولویت آن به مرور زمان افزایش پیدا می‌کند؛ مثلا اگر اولویت‌ها از صفر تا 127 شماره‌گذاری شده باشند و یک پردازه با اولویت صفر (کم‌ترین اولویت ممکن) در صف اجرا موجود باشد، پس از مدت مشخصی، شماره اولویت را یک واحد افزایش می‌دهیم. این کار تا زمانی که پردازنده به پردازه اختصاص یابد، ادامه می‌یابد. علاوه بر آن، راهکارهای زیر به منظور جلوگیری از اختصاص بیش از حد پردازنده به وقفه‌ها نیز در برخی سیستم‌ها اجرا شده‌اند:

* کوتاه و سریع نگه داشتن مدیریت وقفه‌ها؛ برای مثال استفاده از FLIH و SLIH که در بالاتر توضیح داده شد، راهکار مناسبی برای این مورد می‌باشد.
* محدود کردن نرخ ایجاد وقفه‌ها
* کم کردن worst-case نرخ ایجاد وقفه‌ها؛ این کار در سطح دستگاه‌هایی که باعث ایجاد وقفه می‌شوند صورت می‌گیرد.
* چک کردن دوره‌ای اتفاقات[[8]](#footnote-8) بجای استفاده از وقفه‌ها

# زمان‌بندی بازخوردی چند سطحی

در ابتدا یک استراکت به نام schedinfo به فیلدهای استراکت proc اضافه کردیم تا تمامی متغیرهای مربوط به زمان‌بندی یک پردازه در این استراکت قرار بگیرد. یکی از فیلدهای این استراکت، یک enum به نام schedqueue است که نشان می‌دهد پردازه مربوطه در کدام صف قرار دارد.

برای بخش aging تابع زیر نوشته شد:

void  
ageprocs(int osTicks)  
{  
 struct proc \*p;  
 acquire(&ptable.lock);  
  
 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)  
 {  
 if (p->state == RUNNABLE && p->sched\_info.queue != ROUND\_ROBIN)  
 {  
 if (osTicks - p->sched\_info.last\_run > AGING\_THRESHOLD)  
 {  
 release(&ptable.lock);  
 change\_queue(p->pid, ROUND\_ROBIN);  
 acquire(&ptable.lock);  
 }  
 }  
 }  
  
 release(&ptable.lock);  
}

این تابع پس از هر بار افزایش مقدار ticks در interrupt مربوط به تایمر در فایل trap.c، فراخوانی می‌شود. لازم به ذکر است که مقدار p->sched\_info.last\_run پس از هر بار زمان‌بندی در تابع scheduler برای پردازه‌ای که قرار است اجرا شود، مقداردهی می‌شود.

## زمان‌بند نوبت گردشی

برای این زمان‌بند تابع زیر را به سیستم عامل اضافه کردیم. لازم به ذکر است که پارامتر lastScheduled در واقع آخرین پردازه‌ای است که توسط الگوریتم Round-Robin زمان‌بندی شده و این الگوریتم، از پردازه بعد از lastScheduled در صف، به دنبال پردازه‌ای RUNNABLE می‌گردد و در صورت یافتن همچین پردازه‌ای، آن را زمان‌بندی می‌کند.

struct proc\*  
roundrobin(struct proc \*lastScheduled)  
{  
 struct proc \*p = lastScheduled;  
 for (;;)  
 {  
 p++;  
 if (p >= &ptable.proc[NPROC])  
 p = ptable.proc;  
  
 if (p->state == RUNNABLE && p->sched\_info.queue == ROUND\_ROBIN)  
 return p;  
  
 if (p == \*lastScheduled)  
 return 0;  
 }  
}

## زمان‌بند بخت آزمایی

تابع lottery برای پیاده‌سازی این زمان‌بند ایجاد شده است. این تابع ابتدا یک عدد تصادفی ایجاد کرده و سپس در جدول پردازه‌ها (ptable.proc) به دنبال پردازه‌‌هایی می‌گردد که در صف LOTTERY و دارای حالت RUNNABLE می‌باشد. هر پردازه دارای یک مقدار tickets\_count بوده که نشان‌دهنده تعداد بلیط‌های تخصیص یافته به آن پردازه است. برای آنکه پردازه برنده مشخص شود، ابتدا مجموع تعداد بلیط‌ها محاسبه می‌شود و یک عدد تصادفی بین صفر و این مجموع تولید می‌شود. سپس در هر مرحله یک متغیر به نام prev\_interval\_begin را در نظر می‌گیریم که نشان‌دهنده تعداد بلیط‌ها تا آن مرحله از جستجو می‌باشد. در صورتی که عدد تصادفی بین prev\_interval\_begin و آن به علاوه تعداد ticket پردازه کنونی باشد، آن پردازه انتخاب می‌شود. به عبارتی دیگر از جمع انباشته بلیط‌ها برای انجام بخت‌آزمایی استفاده شده است. کد این تابع به صورت زیر می‌باشد:

struct proc\*  
lottery() {  
 struct proc\* result = 0;  
  
 uint tickets\_sum = 0;  
 for (int i = 0; i < NPROC; ++i) {  
 if ((ptable.proc[i].state == RUNNABLE) && (ptable.proc[i].sched\_info.queue == LOTTERY)) {  
 tickets\_sum += ptable.proc[i].sched\_info.tickets\_count;  
 }  
 }  
 if (tickets\_sum == 0)  
 return result;  
  
 uint ticket = rand() % tickets\_sum;  
  
 uint prev\_interval\_begin = 0;  
 for (int i = 0; i < NPROC; ++i) {  
 if (ptable.proc[i].state != RUNNABLE)  
 continue;  
 if (ptable.proc[i].sched\_info.queue != LOTTERY)  
 continue;  
 if (  
 (ticket >= prev\_interval\_begin) &&  
 (ticket <= prev\_interval\_begin + ptable.proc[i].sched\_info.tickets\_count)  
 ) {  
 result = &ptable.proc[i];  
 break;  
 }  
 prev\_interval\_begin += ptable.proc[i].sched\_info.tickets\_count;  
 }  
  
 return result;  
}

## زمان‌بند اول بهترین کار

تابع bestjobfirst این نوع زمان‌بندی را انجام می‌دهد. برای پیاده‌سازی، ابتدا فیلدهای 3 معیار و ضریب متناظر با هر کدام به پردازه اضافه شده است. فیلد priority در اینجا عددی از 1 تا 5 است که نشان دهندۀ اولویت آن پردازه است (1 بالاترین اولویت). در تابع زمان‌بند، بین همه پردازه‌های runnable که نوع صف آنها BJF است، Rank کمینه حساب می‌شود و آن به عنوان پردازه برای اجرا انتخاب می‌شود.

static float  
bjfrank(struct proc\* p)  
{  
 return p->sched\_info.bjf.priority \*  
 p->sched\_info.bjf.priority\_ratio +  
 p->sched\_info.bjf.arrival\_time \*  
 p->sched\_info.bjf.arrival\_time\_ratio +  
 p->sched\_info.bjf.executed\_cycle \*  
 p->sched\_info.bjf.executed\_cycle\_ratio;  
}  
  
struct proc\*  
bestjobfirst(void)  
{  
 struct proc\* result = 0;  
 float minrank;  
  
 struct proc\* p;  
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
 if(p->state != RUNNABLE || p->sched\_info.queue != BJF)  
 continue;  
 float rank = bjfrank(p);  
 if(result == 0 || rank < minrank){  
 result = p;  
 minrank = rank;  
 }  
 }  
  
 return result;  
}

# فراخوانی‌های سیستمی

## تغییر صف پردازه

این فراخوانی سیستمی با نام change\_scheduling\_queue به سیستم عامل اضافه شده است. تابع نهایی مربوط به این فراخوانی سیستمی (در فایل proc.c) در بخش زیر قابل مشاهده است:

int  
change\_queue(int pid, int new\_queue) {  
 struct proc \*p;  
 int old\_queue = -1;  
 if (new\_queue == UNSET)  
 {  
 if (pid == 1)  
 new\_queue = ROUND\_ROBIN;  
 else if (pid > 1)  
 new\_queue = LOTTERY;  
 else  
 return -1;  
 }  
  
 acquire(&ptable.lock);  
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
 if(p->pid == pid){  
 old\_queue = p->sched\_info.queue;  
 p->sched\_info.queue = new\_queue;  
 release(&ptable.lock);  
 return old\_queue;  
 }  
 }  
 release(&ptable.lock);  
 return old\_queue;  
}

## مقداردهی بلیت بخت آزمایی

این فراخوانی سیستمی با نام set\_lottery\_tickets پیاده‌سازی شده است. کد مربوط به آن در زیر قابل مشاهده است:

int  
set\_lottery\_ticket(int pid, int tickets) {  
  struct proc\* p;  
  
  acquire(&ptable.lock);  
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {  
    if ((p->pid == pid) && (p->sched\_info.queue == LOTTERY)) {  
      p->sched\_info.tickets\_count = tickets;  
      release(&ptable.lock);  
      return 1;  
    }  
  }  
  release(&ptable.lock);  
  return -1;  
}

## مقداردهی پارامتر BJF در سطح پردازه

تابع سیستمی این کار با نام set\_bjf\_params\_process که به عنوان ورودی PID پردازه مورد نظر و 3 ضریب BJF را می‌گیرد مطابق کد زیر تعریف شده است:

int  
set\_bjf\_params\_process(int pid, float priority\_ratio, float arrival\_time\_ratio, float executed\_cycles\_ratio)  
{  
 acquire(&ptable.lock);  
 struct proc\* p;  
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
 if(p->pid == pid){  
 p->sched\_info.bjf.priority\_ratio = priority\_ratio;  
 p->sched\_info.bjf.arrival\_time\_ratio = arrival\_time\_ratio;  
 p->sched\_info.bjf.executed\_cycle\_ratio = executed\_cycles\_ratio;  
 release(&ptable.lock);  
 return 0;  
 }  
 }  
 release(&ptable.lock);  
 return -1;  
}

## مقداردهی پارامتر BJF در سطح سیستم

تابع سیستمی این کار با نام set\_bjf\_params\_system که به عنوان ورودی 3 ضریب BJF را می‌گیرد و آن را برای همه پردازه‌ها تنظیم می‌کند، مطابق کد زیر تعریف شده است:

void  
set\_bjf\_params\_system(float priority\_ratio, float arrival\_time\_ratio, float executed\_cycles\_ratio)  
{  
 acquire(&ptable.lock);  
 struct proc\* p;  
 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
 p->sched\_info.bjf.priority\_ratio = priority\_ratio;  
 p->sched\_info.bjf.arrival\_time\_ratio = arrival\_time\_ratio;  
 p->sched\_info.bjf.executed\_cycle\_ratio = executed\_cycles\_ratio;  
 }  
 release(&ptable.lock);  
}

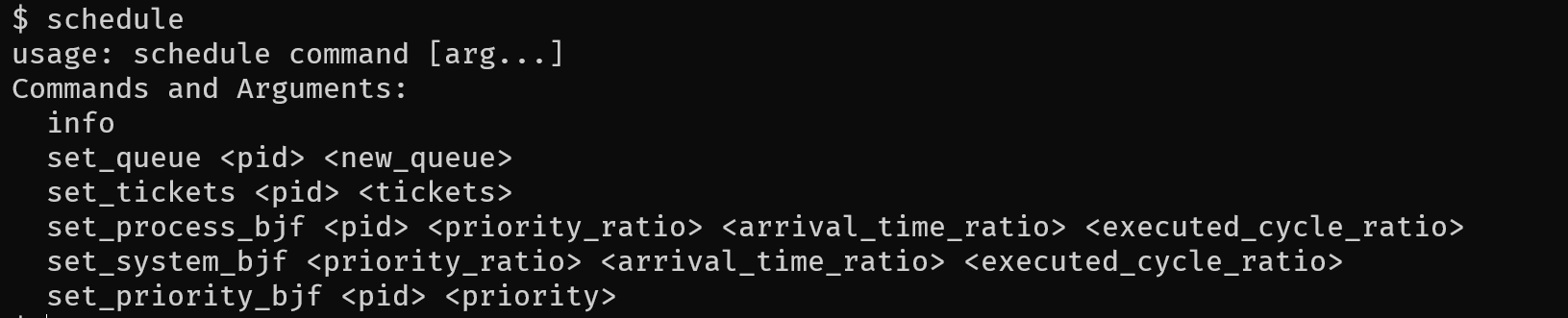
## چاپ اطلاعات

در این فراخوانی سیستمی، روی همه پردازه‌ها پیمایش می‌شود و همه مشخصات خواسته شده پرینت می‌شوند:

void  
print\_process\_info()  
{  
 static char \*states[] = {  
 [UNUSED] "unused",  
 [EMBRYO] "embryo",  
 [SLEEPING] "sleeping",  
 [RUNNABLE] "runnable",  
 [RUNNING] "running",  
 [ZOMBIE] "zombie"  
 };  
  
 static int columns[] = {16, 8, 9, 8, 8, 8, 8, 9, 8, 8, 8, 8};  
 cprintf("Process\_Name PID State Queue Cycle Arrival Ticket Priority R\_Prty R\_Arvl R\_Exec Rank\n"  
 "------------------------------------------------------------------------------------------------------\n");  
  
 struct proc \*p;  
 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 = "???";  
  
 cprintf("%s", p->name);  
 printspaces(columns[0] - strlen(p->name));  
 ...

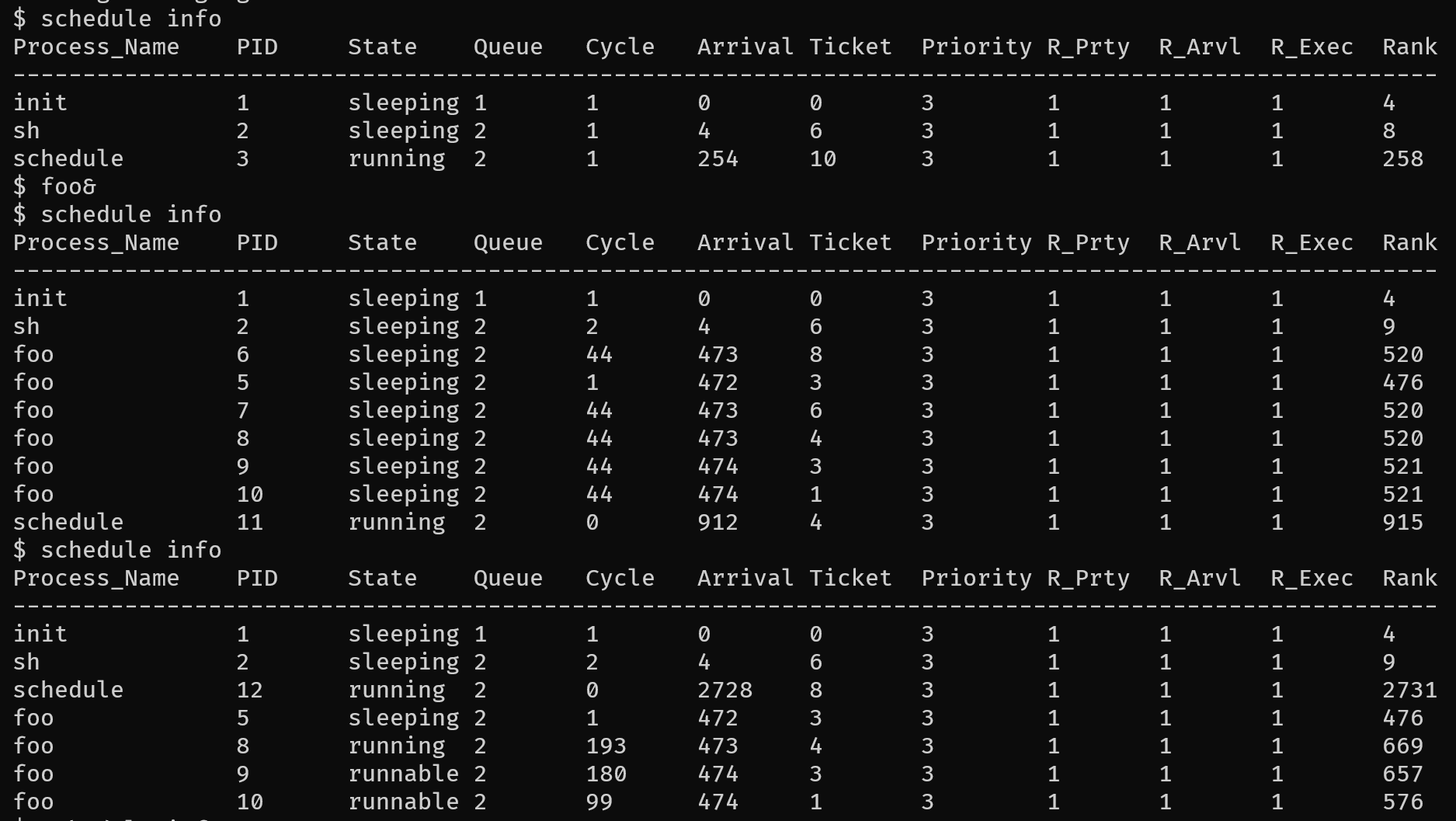
# برنامه سطح کاربر

در ابتدا یک برنامه سطح کاربر برای اجرای فراخوانی‌های سیستمی نوشته شده ایجاد کردیم. نحوه استفاده از این برنامه سطح کاربر به نام schedule به شکل زیر است:



برنامه سطح کاربر دیگری به نام foo نیز نوشته شده که 5 پردازه می‌سازد که هر کدام پس از مدتی sleep کردن، شروع به انجام محاسباتی طولانی می‌کنند.

#include "types.h"  
#include "user.h"  
  
#define PROCS\_NUM 5  
  
int main()  
{  
 for (int i = 0; i < PROCS\_NUM; ++i)  
 {  
 int pid = fork();  
 if (pid > 0)  
 continue;  
 if (pid == 0)  
 {  
 sleep(5000);  
 for (int j = 0; j < 100 \* i; ++j)  
 {  
 int x = 1;  
 for (long k = 0; k < 1000000000000; ++k)  
 x++;  
 }  
 exit();  
 }  
 }  
 while (wait() != -1)  
 ;  
 exit();  
}



1. First-Level Interrupt Handler [↑](#footnote-ref-1)
2. Second-Level Interrupt Handler [↑](#footnote-ref-2)
3. Upper half [↑](#footnote-ref-3)
4. Lower half or bottom half [↑](#footnote-ref-4)
5. Context switch [↑](#footnote-ref-5)
6. Jitter [↑](#footnote-ref-6)
7. Mask [↑](#footnote-ref-7)
8. Polling for events [↑](#footnote-ref-8)