آزمایشگاه سیستم عامل

پروژه سه

اعضای گروه:

پارسا علیزاده - 810101572

نیلوفر مرتضوی - 220701096

محمدرضا خالصی - 810101580

Repository: https://github.com/pars1383/OS-xv6-public

.1

```
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)
};
```

۱. process state = state

۲. process number = pid

۳. Memory limits = sz

۴. list of open file = ofile

۵. program counter = context-> eip

.2

Runnable -> ready

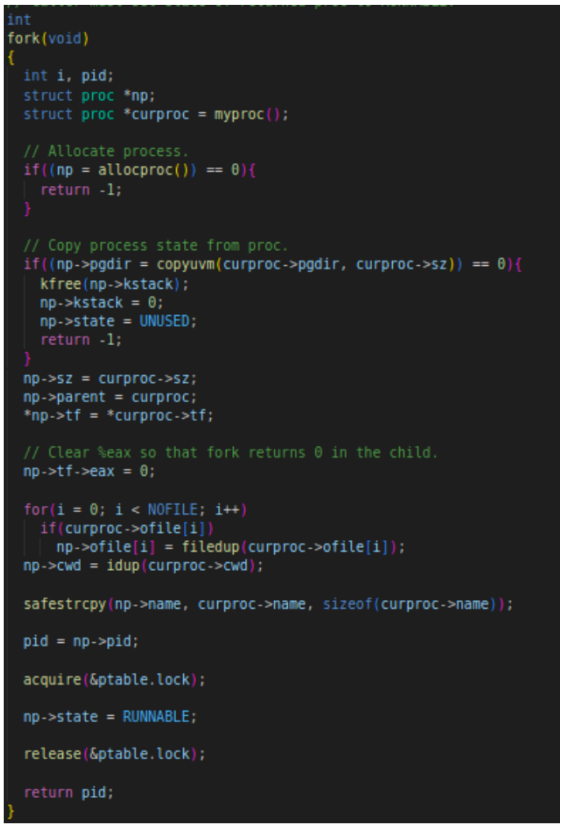
Running -> running

Sleeping -> waiting

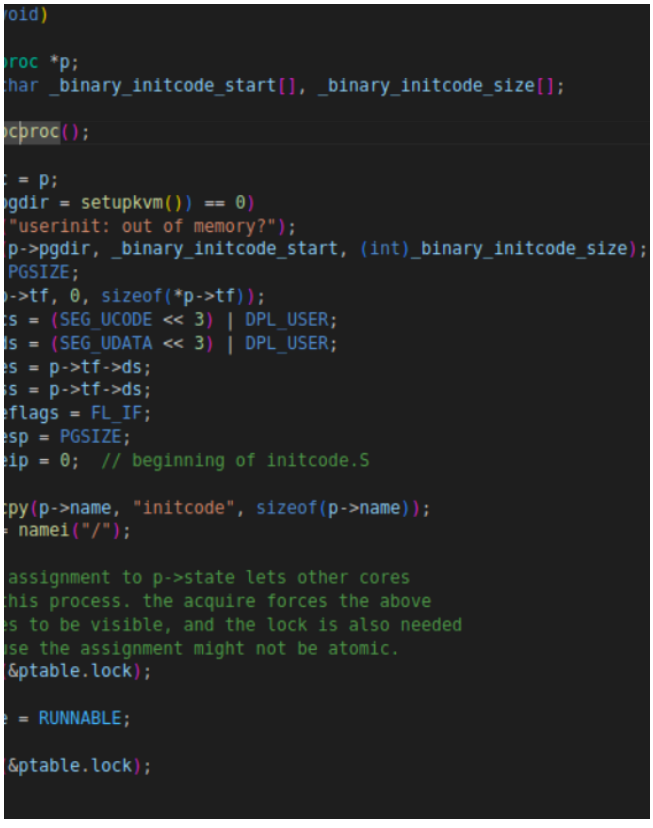
Zombie -> terminate

Embryo -> new

.3



```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```



```
oid)

roc *p;
har _binary_initcode_start[], _binary_initcode_size[];

cproc();

 = p;
gdir = setupkvm()) == 0)
"userinit: out of memory?");
p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
PGSIZE;
->tf, 0, sizeof(*p->tf));
s = (SEG_UCODE << 3) | DPL_USER;
s = (SEG_UDATA << 3) | DPL_USER;
s = p->tf->ds;
s = p->tf->ds;
flags = FL_IF;
sp = PGSIZE;
ip = 0;  // beginning of initcode.S

py(p->name, "initcode", sizeof(p->name));
= namei("/");

assignment to p->state lets other cores
his process. the acquire forces the above
s to be visible, and the lock is also needed
se the assignment might not be atomic.
&ptable.lock);

 = RUNNABLE;

&ptable.lock);
```

اگر پردازه init باشد توسط userinit این کار صورت میگرد در غیر این صورت اینکار توسط تابع fork انجام خواهد گرفت.

در ابتدا پردازه به حالت EMBRYO رفته و سپس به حالت runnable می رود دقیقا مثل همان چیزی که در شکل کتاب گفته شده است.

.4

```
#define NPROC        64  // maximum number of processes
#define KSTACKSIZE 4096  // size of per-process kernel stack
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
#define ROOTDEV       1  // device number of file system root disk
#define MAXARG       32  // max exec arguments
#define MAXOPBLOCKS  10  // max # of blocks any FS op writes
#define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
#define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
#define FSSIZE       1000  // size of file system in blocks
```

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

تعداد پردازه های قابل خلق به اندازه NPROC است که در اینجا ۶۴ است

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

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

  release(&ptable.lock);
  return 0;
```

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
```

در این حالت در آرایه ای از proc ها که در ptable نگه داری میشود استیت اولیه همه پردازه ها UNSEND است و وقتی پردازه جدیدی ایجاد میشود در ایندکس بعد جایی که یکی از عناصر آرایه runnable است پردازه جدید نیز اضافه شده و به حالت runnabble می رود.حال چون تا ایندکس ۶۴ حلقه ادامه پیدا میکند اگر نتواند در آن آرایه جای خالی پیدا کند(به ان معنا که همه ۶۴ پردازه در حالت runnable باشند) تابع allocproc صفر بر میگرداند و اگر صفر برگرداند تابع fork نز با برگرداندن عدد ۱- به کاربر اطلاع میدهد که نمیتواند پردازه جدید ایجاد کند.

5. اگر قبل از حلقه ptable را lock نکنیم ممکن است بصورت concurrent توسط kernel مورد دسترسی قرار بگیرد و دیتای آن عوض شود یا اینکه ممکن است حن schedule پردازه جدیدی وارد شود و در صورتی که نباید در زمانبندی شرکت داشه باشد شرکت داده میشود. همچنن ممکن است با قسمت های دیگه که با این داده کار می کنند و می خواهند روی آن کار کنند همروندی رخ دهد و مشکالتی در داده به وجود بیاورد.

6. در iteration بعدی، چون ساختار داده ptable توسط برنامه scheduler قفل شده است.

7.

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

.8

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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);

  }
```

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

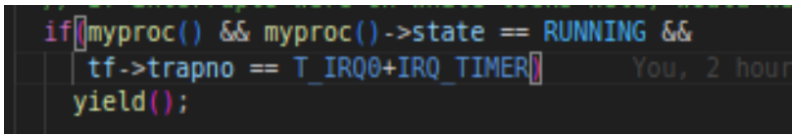
Context Switch for PC-> eip | switch.S

محتوای ثبات های قبلی را در پشه ذخیره می کند و محتوای ثبات های جدید را از پشه به داخل ثبات ها می آورد.

9. عمالا دیگر وقفه در برنامه ریزی موثر نبود و اجرای برنامه ها به صورت FIFO انجام می شد و دیگر اشتراک زمانی در بین آنها نداشتیم.
   a. عدم پاسخ به رویدادهای ورودی/خروجی، که باعث توقف پردازه‌های وابسته به I/O می‌شود.
   b. خطر قفل سیستم در صورت نبود پردازه‌های RUNNABLE.
   c. کاهش کارایی و قابلیت اطمینان کلی سیستم.

10. وقفه تایمر در xv6 هر 10 میلی‌ثانیه صادر می‌شود. این نتیجه بر اساس تنظیمات تابع lapicinit محاسبه شده است، که رجیستر شمارش اولیه تایمر (TICR) را روی 10,000,000 تنظیم می‌کند و با فرض فرکانس کلاک 1 گیگاهرتز، فاصله زمانی 10 میلی‌ثانیه به دست می‌آید. این فرکانس برای زمان‌بندی پیش‌خرید پردازه‌ها و مدیریت زمان در xv6 استفاده می‌شود.

11. تابع trap این کار را انجام میدهد.



12.

زمان کوانتوم ما باید بیشتر از زمان intrupt در سیستم باشد

Q>10

Q = 12 ms

13. تابع wait در نهایت از تابع sleep استفاده می کند تا اجرای برنامه را ا زمان اتمام کار فرآیند فرزند متوقف کند. فرآیند والد، تابع sleep را فراخوانی می کند تا منتظر دریافت سیگنالی از فرزند باشد که نشاندهنده

پایان کار آن است. این مکانیسم از انتظار مشغول (waiting busy) جلوگیری می کند، که در آن والد به طور مداوم وضعیت فرزند را بررسی می کند و باعث مصرف غیرضروری سیکل های CPU میشود.

14. فراتر از استفاده آن در تابع wait، تابع sleep در سناریوهایی کاربرد دارد که یک فرآیند نیاز به توقف تا زمان برآورده شدن یک شرط خاص دارد. یکی از این موارد، هماهنگسازی خوانندهها و نویسندهها در pipe ها است. هنگای که بافر pipe پر می شود، فرآیند نویسنده نابع sleep را فراخوانی می کند نا منتظر بماند تا یک خواننده مقداری از داده ها را مصرف کند و فضای در بافر آزاد شود.

15. در کرنل 6xv، نابعی که مسئول آگاهسازی یک فرآیند درباره رویدادی است که منتظر آن بوده، نابع wakeup است. هنگای که event مورد انتظار رخ می دهد، تابع wakeup مربوطه فراخوانی می شود، فرآیند waiting را به عنوان RUNNABLE علالمت گذاری می کند و به آن اجازه می دهد تا اجرای خود را از سر بگیرد.

16. برنامه را از وضعیت waiting به وضعیت ready منتقل می کند.

17. بله، تابع دیگری که می تواند باعث اننقال از حالت انتظار به حالت آماده شود، نابع yield است. یک فرآیند ممکن است داوطلبانه با استفاده از نابع yield، پراسسور را رها کند. به عنوان مثال، فرآیندی که محاسبات طولانی انجام می دهد، می تواند به طور دورهای yield را فراخوانی کند تا به سایر فرآیندها اجازه اجرا دهد. ابن کار تخصیص عادلانه CPU را تضمین می کند و از انحصار پردازنده توسط یک فرآیند واحد جلوگری می کند. تابع yield به طور موثر فرآیند را از حالت RUNNING به RUNNABLE منتقل می کند، که این امکان را فراهم می سازد تا بعداً دوباره زمانبندی شود.

18. در 6xv، زمانی که یک فرآیند والد خاتمه می یابد و فرآیندهای فرزند خود را به عنوان یتیم باقی می گذارد، فرآیند init این فرآیندهای یتیم را به فرزندی می پذیرد. فرآیند init، که یک فرآیند ویژه سیستی است، به صورت دورهای فراخوانی `wait` را انجام می دهد تا منابع این فرآیندهای یتیم (فرآیندهای زامی) را پاکسازی کند. این اطمینان حاصل میشود که هچ فرآیند زامبی به طور نامحدود در سیستم باقی نمی ماند و از نشت منابع جلوگری می شود.

19. در این سناریو، اگر تمام پردازنده‌ها مشغول اجرای فرآیندهای کلاس اول باشند، پوسته (xv6 (shell هیچ زمانی از CPU دریافت نمی‌کند و به حالت starvation (گرسنگی) دچار می‌شود. این به دلیل اولویت ثابت (fixed priority) در مکانیزم زمان‌بندی است که ابتدا کلاس اول (با اولویت بالاتر) بررسی می‌شود و تنها در صورتی که هیچ فرآیند قابل اجرا (RUNNABLE) در کلاس اول وجود نداشته باشد، زمان‌بندی‌کننده به کلاس دوم و سطوح آن می‌رسد. در نتیجه:
- پوسته که در کلاس دوم، سطح اول قرار دارد، نمی‌تواند اجرا شود و سیستم از نظر کاربری قفل می‌شود (unresponsive).
- کاربر نمی‌تواند دستورات جدیدی وارد کند یا از پوسته استفاده کند، زیرا فرآیند sh برای دریافت ورودی و پردازش آن به CPU نیاز دارد.
- این مسئله یک مشکل جدی برای قابلیت استفاده از سیستم است، به‌ویژه در محیط‌هایی که هم فرآیندهای بلادرنگ و هم تعاملی (مانند پوسته) باید به‌طور همزمان پشتیبانی شوند. از دیدگاه من، این رفتار به دلیل اولویت‌بندی ثابت و عدم وجود مکانیزمی برای تضمین حداقل زمان CPU برای فرآیندهای تعاملی (مانند پوسته) رخ می‌دهد.

20. خیر، با تغییر CPUS به ۲، ترتیب اجرای فرآیندهایی که قبلاً در حالت تک‌پردازنده مشاهده کرده‌اید، پابرجا نخواهد بود. در حالت تک‌پردازنده، به دلیل وجود یک CPU، فرآیندها به‌صورت متوالی و بر اساس الگوریتم RR اجرا می‌شوند و ترتیب اجرای آن‌ها کاملاً قابل پیش‌بینی است. اما در حالت دوپردازنده‌ای،

هر CPU می‌تواند به‌طور مستقل فرآیندهای RUNNABLE را از صف‌های مختلف (کلاس ۱، کلاس ۲ سطح ۱، یا کلاس ۲ سطح ۲) انتخاب و اجرا کند، که منجر به تغییر در ترتیب اجرای فرآیندها می‌شود.

ترتیب اجرای فرآیندها در حالت CPUS = 2 به دلیل اجرای موازی توسط دو پردازنده و عدم هماهنگی کامل در انتخاب فرآیندها از صف RR، پابرجا نخواهد بود. این تغییر طبیعی است، زیرا معماری چندپردازنده‌ای بهینه‌سازی‌شده برای استفاده حداکثری از منابع است، اما پیش‌بینی دقیق ترتیب اجرای فرآیندها را دشوار می‌کند. اگر نیاز به حفظ ترتیب خاص در حالت چندپردازنده‌ای باشد، باید مکانیزمی برای همگام‌سازی (synchronization) بین CPUها (مثلاً با استفاده از قفل‌های اضافی) پیاده‌سازی شود، که ممکن است عملکرد را کاهش دهد.

21. برای سیستم‌های Hard Real-Time، فرض پیاده‌سازی مناسب‌تر این است که:

- هنگام ایجاد فرآیند، یک تحلیل امکان‌پذیری انجام شود و فرآیندهایی که نمی‌توانند مهلت خود را رعایت کنند، رد شوند (Admission Control).
- فرآیندهایی که مهلت خود را از دست داده‌اند، فوراً حذف شوند تا از تأثیرگذاری روی سایر فرآیندها جلوگیری شود.
- مدیریت منابع (مانند وقفه‌ها و قفل‌ها) به‌گونه‌ای باشد که تضمین کند هیچ مهلتی از دست نمی‌رود.

این رویکرد تضمین می‌کند که سیستم همیشه در محدوده مهلت‌های تعیین‌شده عمل کند، که برای سیستم‌های Hard Real-Time حیاتی است. برخلاف Soft Real-Time که تأخیر قابل تحمل است، در Hard Real-Time هرگونه نقض مهلت می‌تواند منجر به خرابی سیستم شود، بنابراین باید با سخت‌گیری بیشتری مدیریت شود.

22. مدت زمانی که فرآیند در وضعیت SLEEPING باشد به‌عنوان زمان انتظار در نظر گرفته نمی‌شود، زیرا این فرآیند در این حالت آماده اجرا نیست و منتظر یک رویداد خارجی است، نه محرومیت از CPU به دلیل زمان‌بندی. محاسبه زمان انتظار فقط برای فرآیندهای RUNNABLE انجام می‌شود که در صف زمان‌بندی هستند و ممکن است به دلیل اولویت یا کوانتوم اجرا نشوند. این رویکرد از تحریف اولویت‌ها جلوگیری کرده و با هدف مکانیزم‌هایی مانند aging (که برای رفع گرسنگی طراحی شده) سازگار است.

23. برای مدیریت تعداد فرآیندهای RUNNABLE در هر صف، باید متغیرهای runnable_class1، runnable_class2_level1، و runnable_class2_level2 در توابع زیر به‌روزرسانی شوند:
    - allocproc: هنگام ایجاد فرآیند جدید.
    - exit: هنگام خروج فرآیند.
    - sleep: هنگام انتقال به حالت SLEEPING.
    - wakeup: هنگام بازگشت به حالت RUNNABLE.
    - sys_create_rtproc: هنگام ایجاد فرآیند بلادرنگ در کلاس ۱.
    - sys_change_level: هنگام تغییر سطح در کلاس ۲.
    - sys_exec: هنگام تنظیم سطح فرآیندهای exec شده.
    - trap: هنگام انتقال فرآیندها در مکانیزم aging.
    - sys_fork: هنگام تنظیم سطح فرزندان پوسته.

این توابع تمام نقاطی را که حالت فرآیند یا صف آن تغییر می‌کند، پوشش می‌دهند و با استفاده از قفل ptable.lock، از مشکلات هم‌زمانی (race condition) جلوگیری می‌شود.