الناز رضایی – حوریه سبزواری **شماره دانشجویی:98411387 - 98412004**

گزارش بخش دوم فاز دوم پروژه xv6:

ابتدا در فایل proc.h ساختار هر process را تغییر می‌دهیم و تعدادی field به آن اضافه می‌کنیم.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  ...
  int stime;                   // Process strat time
  int rtime;                   // Process run time
  int iotime;                  // Process I/O time
  int etime;                   // Process end time
  int priority;                // Process Priority for priority scheduling
};
```

همان طور که در کامنت‌ها ذکر شده، stime زمان شروع هر process، rtime زمان اجرای هر process، iotime همان مجموع زمان مربوط به IO، etime زمان خاتمه process و priority اولویت اجرای فرآیند است.

بعد از این نوبت به تعریف فراخوانی سیستمی (system call) ای به نام waitx می‌رسد.

برای این کار ابتدا در ***فایل syscall.h*** یک سری شماره جدید تعریف می‌کنیم. از این شماره در آرایه‌ای که قرار است برای routine پاسخگویی به آن فراخوانی سیستمی انجام می‌دهیم.

```
// new system call number
#define SYS_waitx  22
#define SYS_set_priority  23
#define SYS_yield  24
#define SYS_get_pid  25
```

آخرین شماره‌ای که در این فایل تعریف شده 21 می‌باشد. برای راحتی کار ما از عدد بعدی(22) برای فراخوانی سیستمی جدید استفاده می‌کنیم.

سپس در ***فایل syscall.c*** خط زیر را باید اضافه کنیم: (این خط برای تعریف routine پاسخگویی به آن فراخوانی سیستمی waitx است، یعنی هر گاه که برنامه سطح کاربر از سیستم عامل درخواست کرد، این تابع برای پاسخگویی به آن اجرا می‌شود)

```
// new system call routine
extern int sys_waitx(void);
extern int sys_set_priority(void);
```

```
extern int sys_yield(void);
extern int sys_get_pid(void);
```

این خط صرفاً تعریف یا signature تابع را صدا می‌کنیم.

سپس در آرایه‌ای که در آن اشاره‌گر به تابع‌هایی که routine های پاسخگویی به فراخوانی‌های سیستمی را قرار داده ایم، موردی برای فراخوانی سیستمی جدید را اضافه می‌کنیم. (این آرایه تمام routine های پاسخگویی به فراخوانی های سیستمی را در خود جای داده است)

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
...
// new system call routine function pointer
[SYS_waitx]   sys_waitx,
[SYS_set_priority]      sys_set_priority,
[SYS_get_pid] sys_get_pid,
[SYS_yield]   sys_yield,
};
```

سپس خطوط زیر را در ***فایل user.h*** زیر جایی که آخرین system call تعریف شده است اضافه می‌کنیم: (این قسمت signature تابع سیستمی جدید در سطح kernel است)

```
// system calls
int fork(void);
...
// new system call function
int waitx(int*, int*);
int set_priority(int,int);
int get_pid(void);
int yield(void);
```

سپس در ***فایل usys.S*** نیز اسم تابع سیستمی خود را تعریف می‌کنیم.

SYSCALL(fork)

...

SYSCALL(waitx)

SYSCALL(set_priority)

SYSCALL(get_pid)

SYSCALL(yield)

سپس در ***فایل sysproc.c*** یک system process برای پاسخگویی به interrupt مربوط به فراخوانی‌های سیستمی را تعریف می‌کنیم: (منظور همان روتین پاسخگویی به فراخوانی سیستمی است)

```
int sys_waitx(void)
{
    int *wtime, *rtime;
    if (argptr(0, (void*)&wtime, sizeof(wtime)) < 0)
        return -1;
    if (argptr(1, (void*)&rtime, sizeof(rtime)) < 0)
        return -1;
    return waitx(wtime, rtime);
}

int sys_set_priority(void)
{
    int pid,priority;
    if(argint(0, &pid) < 0)
        return -1;
    if(argint(1,&priority) < 0)
        return -1;
    return set_priority(pid,priority);
}

int sys_get_pid(void)
{
    return myproc()->pid;
}
```

در این قسمت ورودی ها را خوانده و آن ها را به متغیر های خود که می‌خواهیم آنها را تغییر دهیم نگاشت می‌کنیم (برای این کار از دستورات argint استفاده کرده ایم که ورودی را از آرگومان‌های فراخوانی سیستمی می‌خوانند، استفاده کرده ایم.) .

در ***فایل defs.h*** نیز در قسمت مربوط به proc.c// خط signature تابع را تعریف می‌کنیم:

```
//PAGEBREAK: 16
// proc.c
int             cpuid(void);
...
// new function for user level
int             waitx(int*, int*);
int             set_priority(int,int);
```

```
int             get_pid(void);
```

در ***فایل proc.c*** نیز تعریف توابع را اضافه می‌کنیم:

```
int
waitx(int *wtime, int *rtime)
{
    struct proc *p;
    int havekids, pid;
    struct proc *curproc = myproc();

    acquire(&ptable.lock);
    for(;;){
        // Scan through table looking for exited children.
        havekids = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->parent != curproc)
                continue;
            havekids = 1;
            if(p->state == ZOMBIE){
                // Found one.
                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                freevm(p->pgdir);
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                p->state = UNUSED;
                *wtime=p->etime-p->stime-p->rtime;
                *rtime=p->rtime;
                release(&ptable.lock);
                return pid;
            }
        }

        // No point waiting if we don't have any children.
        if(!havekids || curproc->killed){
            release(&ptable.lock);
            return -1;
        }

        // Wait for children to exit.  (See wakeup1 call in proc_exit.)
```

```
        sleep(curproc, &ptable.lock);  //DOC: wait-sleep
    }
}
```

در این قسمت در ابتدا همه پردازه ها را چک کرده و در صورتی که یک پردازه پیدا کردیم که parent آن پردازه فعلی در حال اجرا باشد، اطلاعات خواسته شده در فراخوانی سیستمی را باز می‌گردانیم.

برای آپدیت نگه داشتن زمان‌ها برای هر فرآیند، تابع زیر را می‌نویسیم (در فایل proc.c) :

```
void updateProcessTimes()
{
    struct proc *p;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        switch(p->state) {
            case SLEEPING:
                p->iotime++;
                break;
            case RUNNABLE:
                p->iotime++;
                break;
            case RUNNING:
                p->rtime++;
                break;
            default:
                ;
        }
    }
    release(&ptable.lock);
}
```

سپس باید زمان بند xv6 را تغییر دهیم:

```
//for part 2
void
scheduler(void)
{
   struct proc *p = 0;

   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++)
      {

          struct proc *maxPriority = 0;
          struct proc *p1 = 0;

          if(p->state != RUNNABLE)
              continue;
          // Choose the process with highest priority among RUNNABLE processes
          maxPriority = p;
          for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)
          {
              if((p1->state == RUNNABLE) && (maxPriority->priority > p1->priority))
              {
                  maxPriority = p1;
              }
          }

          if(maxPriority != 0)
              p = maxPriority;

          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);
  }
}
```

و سپس در فایل trap.c در تابع trap، در جایی که clock یا همان ticks را آپدیت می‌کنیم، آن را صدا می‌کنیم:

```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    ...
      ticks++;
      //---------------------------------------------------------------
      //updating times for the process
      updateProcessTimes();

      //---------------------------------------------------------------
       ...
}
```

برای تست برنامه نوشته شده نیز یک فایل جدید ایجاد می‌کنیم تا بتوانیم دستور جدیدی که می‌خواهیم در کرنل تعریف کنیم را در آن قرار دهیم و آن را تست کنیم.

پس یک فایل جدید به نام prs.c ایجاد می‌کنیم و قطعه کد زیر را در آن قرار می‌دهیم:

```
#include "types.h"
#include "stat.h"
#include "user.h"

int wx(int* wtime, int* rtime)
{
    return waitx(wtime,rtime);
}

int main(int argc, char *argv[])
{
    int a = fork();

    if(a>0){
        int npid=fork();
        if(npid==0){
            malloc( 1710 * (sizeof(int)));
            int tmp=set_priority(get_pid(),50);
            yield();
            printf(1,"%d\n",tmp);
            exit();
        }
```

```
        int nnpid=fork();
        if(nnpid==0){
            malloc(1750 * (sizeof(int)));
            int tmp=set_priority(get_pid(),51);
            yield();
            printf(1,"%d\n",tmp);

            exit();
        }

//        wx(&wtime,&rtime);
//        printf(1,"\n*************in user level************\n");
//        printf(1,"wait time= %d, run time= %d\n ",wtime,rtime);
//        printf(1,"\n");
        wait();
        wait();
        wait();
        wait();
//        wx(&wtime,&rtime);
//        printf(1,"wait time= %d, run time= %d\n ",wtime,rtime);
//        wx(&wtime,&rtime);
//        printf(1,"wait time= %d, run time= %d\n ",wtime,rtime);
//        wx(&wtime,&rtime);
//        printf(1,"wait time= %d, run time= %d\n ",wtime,rtime);

        exit();
    }
    else
    {
        if(fork()){
            malloc(1700 * (sizeof(int)));
            if (fork())
            {
                malloc(1008 * (sizeof(int)));
                int tmp=set_priority(get_pid(),53);
                yield();
                printf(1,"%d\n",tmp);

                wait();
                wait();
                wait();

                exit();
            }
```

```
            else
            {
                malloc(1800 * (sizeof(int)));
                int tmp=set_priority(get_pid(),54);
                yield();
                printf(1,"diff:%d\n",tmp);
                tmp=set_priority(get_pid(),42);
                yield();
                printf(1,"diff2:%d\n",tmp);

                //-----------------------------------------------------------------
-----------------------------
                exit();
            }


        }else{
            malloc(1090 * (sizeof(int)));
            if (fork())
            {
                malloc( 1030 * (sizeof(int)));
                int tmp=set_priority(get_pid(),55);
                yield();
                printf(1,"%d\n",tmp);
                wait();

                wait();
                exit();
            }
            else
            {
                malloc(1100 * (sizeof(int)));
                exit();
            }
        }

    }
}
```

در قطعه کد بالا تعدادی پردازه جدید ایجاد کرده (زیرا تعداد پردازه‌های خود سیستم عامل xv6 زیاد نیست) و سپس فراخوانی سیستمی که آن را در تابع set_priority و سپس yield را برای rescheduling صدا می‌کنیم. به خاطر اینکه در سطح user هستیم از دستور printf استفاده کرده ایم که آرگومان اول آن جایی است که باید آرگومان دوم در آن نوشته شود. ما نیز 1 را که همان stdout است به تابع پاس داده‌ایم و این یعنی آرگومان دوم را روی کنسول نمایش می‌دهد.

در آخر نیز برای کامپایل کردن کد و نیز لینک کردن object های ایجاد شده توسط کامپایلر، خطوط زیر را باید در قسمت های گفته شده به MakeFile موجود در پروژه اضافه کنیم:

```
UPROGS=\
    _cat\
    ...
    _wx\
    _prs\
```

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ...
    Wx.c\
    prs.c\
```

سپس برای امتحان کردن برنامه نوشته شده باید دستورات زیر را اجرا کنیم:

make clean

make qemu

بعد از اجرای این دستورات shell مربوط به سیستم عامل xv6 باز می‌شود و ما می‌توانیم دستور جدید و برنامه خود را تست کنیم.

وقتی که shell باز شد باید دستور زیر را در آن وارد کنیم:

prs

نمونه‌هایی از خروجی‌های برنامه در زیر آورده شده است:

```
init: starting sh
$ prs
60
60
60
diff:6600

diff2:54
```

```
$ prs
60
60
diff:60
diff2:54
60
60
```

با صدا کردن فراخوانی های سیستمی خود xv6 هم به مشکلی نخوردیم:

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16388
echo           2 4 15240
forktest       2 5 9544
grep           2 6 18608
init           2 7 15828
kill           2 8 15268
ln             2 9 15124
ls             2 10 17752
mkdir          2 11 15368
rm             2 12 15344
sh             2 13 27980
stressfs       2 14 16260
usertests      2 15 67364
wc             2 16 17120
zombie         2 17 14936
wx             2 18 16976
prs            2 19 17160
console        3 20 0
```