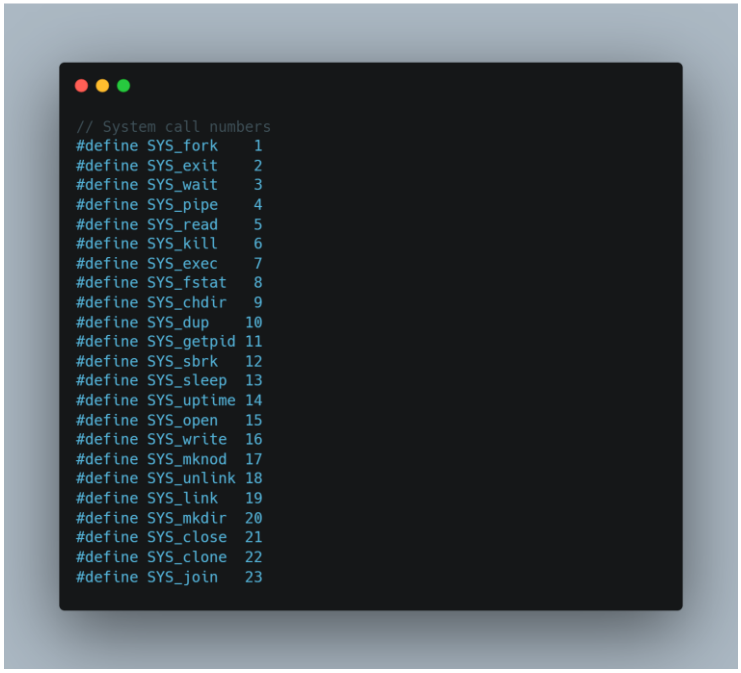
به نام خدا

پروژه پایانی درس سیستمهای عامل

بهاره کاوسی نژاد – 99431217

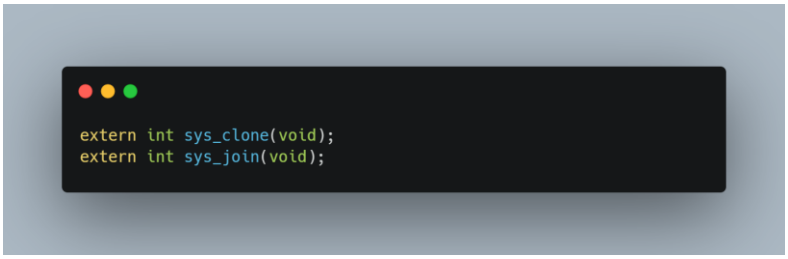
شکیبا انارکی فیروز – 99442047

ابتدا در فایل syscall.h این دو سیستم کال را اضافه کرده و به آنها شماره اختصاص می‌دهیم.



```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_clone  22
#define SYS_join   23
```

این دو سیستم کال را در فایل syscall.c تعریف می‌کنیم:



```
extern int sys_clone(void);
extern int sys_join(void);
```

```
[SYS_clone]   sys_clone,
[SYS_join]    sys_join
```

پیاده سازی در فایل sysproc.c انجام می‌شود؛ ورودی‌های پاس داده شده به توابع را در فایل sysproc.c با استفاده از argint دریافت می‌کنیم.

```c
int sys_clone(void) // system call for cloning - Checking arguments
{
  int functionName, argument1, argument2, stack;
  if(argint(0, &functionName)<0 || argint(1, &argument1)<0 ||
argint(2, &argument2)<0 || argint(3, &stack)<0)
    return -1;
  return clone((void *)functionName, (void *)argument1, (void
*)argument2, (void *)stack);
}

int sys_join(void) // system call for joining
{
  void **joinStack;
  int joinStackArg;
  joinStackArg = argint(0, &joinStackArg);
  joinStack = (void**) joinStackArg;
  return join(joinStack);
}
```

در این سیستم کال‌ها از توابع clone و join استفاده شده است که در ادامه آنها را تعریف می‌کنیم.

به منظور دسترسی کاربر به این سیستم کال‌ها باید در فایل usys.s آنها را به صورت زیر:

```
SYSCALL(clone)
SYSCALL(join)
```

و در user.h آنها را به همراه ورودی‌هایشان تعریف می‌کنیم:

```
// adding clone and join signature
int clone(void (*start_routine)(void*,void*), void *, void *, void *);
int join(void**);
```

تعدادی توابع داریم که قرار است به عنوان library مورد استفاده قرار گیرند؛ این توابع را نیز در فایل user.h تعریف می‌کنیم:

```
// other functions needed:
int thread_create(void (*start_routine)(void *,void*), void * arg1,
void * arg2);
int thread_join();
int lock_init(lock_thread *lk);
void lock_acquire(lock_thread *lk);
void lock_release(lock_thread *lk);
```

همچنین برای استفاده از مفهوم lock در فایل user.h یک struct با نام lock_thread تعریف می‌کنیم که با یک flag عمل lock را انجام می‌دهد:

```
typedef struct __lock_thread{
  uint isLocked;
}lock_thread;
```

پیاده‌سازی توابع clone و join در فایل proc.c انجام شده است و تعریف این دو تابع در فایل defs.h انجام می‌شود:

- تابع clone با اطلاعات stack داده شده به آن یک process جدید می‌سازد.
- تابع join در process table به دنبال child processهایی می‌گردد که به حالت zombie در آمده‌اند تا آن‌ها را از جدول پاک کند.

تابع clone:

در ابتدا با تابع allocproc به یک پراسس جدید حافظه اختصاص می‌دهیم و اگر خروجی آن -۱ باشد یعنی اروری در حین آن صورت گرفته است و در عیر این صورت ۰ بر می گرداند.

سپس دیتا را با استفاده از pgdir مدیریت می‌کنیم و آدرس والد آن را هم می گذاریم.

همچنین trapframe که برای تنظیم کردن cpu برای اجرای ترد لازم است را ایجاد میکنیم و همچینین isthread را یک میکنیم که برای scheduler مورد استفاده قرار خواهد گرفت.

برای ترد جدید استک ایجاد کرده و ورودی های آن را به استک وارد میکنیم. پوینتر های استک و instruction را ایجاد و مقداردهی کرده و سپس وضعیت ترد را به runnable تغیر می دهیم .

همچنین برای والد تعداد childcount را اضافه کرده و pid به ترد اختصاص داده و آن را در آخر بر میگردانیم.

تابع join:

در این تابع به یک حلقه ی بی نهایت نیاز داریم تا همواره چک کنیم که آیا پراسس فرزندی terminate شده است یا نه. داخل حلقه پراسس های ptable بررسی می شوند و ابتدا وجود child thread و سپس zombie هاچک می شود . برای پایان دادن هم ورودی داخل process table و پوینتر های مربوط به آن پاک می شود.

اگر پراسس فرزند نداشته باشد یا تمام شده باشد ، مقدار -۱ برگشت داده می شود و در پایان هم اگر هیچ فرزندی terminate نشده باشد والد آن ها به حالت sleep می رود تا زمان terminate شدن با سیگنالwakeup1 آن را متوجه کنند و به کار خود ادامه دهد.

```c
int clone(void(*fcn)(void*,void*), void *arg1, void *arg2, void*
stack)
{
  struct proc *np;
  struct proc *p = myproc();

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

  // Copy process data to the new thread
  np->pgdir = p->pgdir;
  np->sz = p->sz;
  np->parent = p;
  *np->tf = *p->tf;
  np->IsThread = 1;
  void * sarg1, *sarg2, *sret;

  // Push fake return address to the stack of thread
  sret = stack + PGSIZE - 3 * sizeof(void *);
  *(uint*)sret = 0xFFFFFFF;

  // Push first argument to the stack of thread
  sarg1 = stack + PGSIZE - 2 * sizeof(void *);
  *(uint*)sarg1 = (uint)arg1;

  // Push second argument to the stack of thread
  sarg2 = stack + PGSIZE - 1 * sizeof(void *);
  *(uint*)sarg2 = (uint)arg2;

  // Put address of new stack in the stack pointer (ESP)
  np->tf->esp = (uint) stack;

  // Save address of stack
  np->threadstack = stack;

  // Initialize stack pointer to appropriate address
  np->tf->esp += PGSIZE - 3 * sizeof(void*);
  np->tf->ebp = np->tf->esp;

  // Set instruction pointer to given function
  np->tf->eip = (uint) fcn;

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

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

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

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  p -> childCount++;

  release(&ptable.lock);

  return np->pid;
}
```

```
int
join(void** stack)
{
  struct proc *p;
  int havekids, pid;
  struct proc *cp = myproc();
  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {

      // Check if this is a child thread (parent or shared address space)
      if(p->parent != cp || p->pgdir != p->parent->pgdir)
        continue;

      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;

        // Remove thread from the kernel stack
        kfree(p->kstack);
        p->kstack = 0;

        // Reset thread in process table
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        stack = p->threadstack;
        p->threadstack = 0;

        release(&ptable.lock);
        return pid;
      }
    }

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

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

تعریف توابع clone و join در defs.h:

```c
// syscall.c
int             argint(int, int*);
int             argptr(int, char**, int);
int             argstr(int, char**);
int             fetchint(uint, int*);
int             fetchstr(uint, char**);
void            syscall(void);
int             clone(void(*fcn)(void*, void*), void*, void*, void*);
int             join(void**);
```

در فایل proc.h یک struct برای processها وجود دارد که در آن property های جدیدی تحت عنوان threadstack، turn، childCount و IsThread اضافه می‌کنیم.

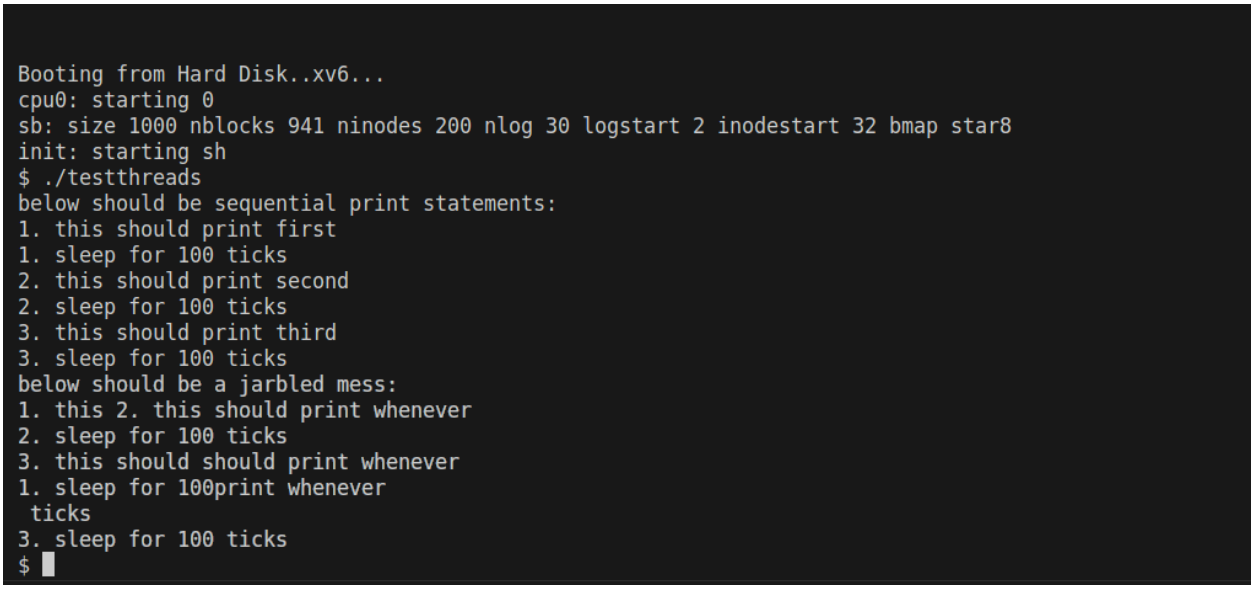
```c
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this
process
  void *threadstack;           // Address of thread stack to be freed
  enum procstate state;        // 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)
  int childCount;
  int turn;                    // Turn of the process
  int IsThread;                // Is this a thread
};
```

تعریف توابع جدید که برای library در نظر گرفتیم باید در فایل ulib.c قرار بگیرند:

```c
// defining functions bodies
int thread_create(void (*start_routine)(void *, void *), void* arg1,
void* arg2)
{
  void* threadStack;
  threadStack = malloc(PGSIZE); // taking memory for a page size

  return clone(start_routine, arg1, arg2, threadStack);
}

int thread_join()
{
  void * stackPtr;
  int x = join(&stackPtr);
  return x;
}

int lock_init(lock_thread *lk)
{
  lk->isLocked = 0;
  return 0;
}

void lock_acquire(lock_thread *lk){
  //prevent interruption .
  //take a pointer to a lock_thread structure as an argument and
returns nothing (void).
  while(xchg(&lk->isLocked, 1) != 0);
}

void lock_release(lock_thread *lk){
  // xchg = exchange
    xchg(&lk->isLocked, 0);
}
```

نمونه تست و نتیجه اجرای آن با دستور make qemu-nox و سپس ./testthreads:

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
#define SLEEP_TIME 10

lock_thread* lk;

void f1(void* argument1, void* argument2) {
  int num = *(int*)argument1;
  if (num) lock_acquire(lk);
  printf(1, "Function 1 is printing: %s\n", num ? "first" : "whenever");
  printf(1, "Function 1 sleep for %d ms\n", SLEEP_TIME);
  sleep(SLEEP_TIME);
  if (num) lock_release(lk);
  exit();
}

void f2(void* argument1, void* argument2) {
  int num = *(int*)argument1;
  if (num) lock_acquire(lk);
  printf(1, "Function 2 is printing: %s\n", num ? "second" : "whenever");
  printf(1, "Function 2 sleep for %d ms\n", SLEEP_TIME);
  sleep(SLEEP_TIME);
  if (num) lock_release(lk);
  exit();
}

void f3(void* argument1, void* argument2) {
  int num = *(int*)argument1;
  if (num) lock_acquire(lk);
  printf(1, "Function 3 is printing: %s\n", num ? "third" : "whenever");
  printf(1, "Function 3 sleep for %d ms\n", SLEEP_TIME);
  sleep(SLEEP_TIME);
  if (num) lock_release(lk);
  exit();
}

int main(int argc, char *argv[])
{
    lock_init(lk);
    int arg1 = 0, arg2 = 1;

    printf(1, "Test 1: Two threads with lock, one without:\n");
    thread_create(&f1, (void *)&arg1, (void *)&arg2); // Thread with lock
    thread_create(&f2, (void *)&arg1, (void *)&arg2); // Thread with lock
    arg1 = 0;
    thread_create(&f3, (void *)&arg1, (void *)&arg2); // Thread without lock
    thread_join();
    thread_join();
    thread_join();

    arg1 = 1;
    printf(1, "Test 2: One thread with lock, two without:\n");
    thread_create(&f1, (void *)&arg1, (void *)&arg2); // Thread with lock
    arg1 = 0;
    thread_create(&f2, (void *)&arg1, (void *)&arg2); // Thread without lock
    thread_create(&f3, (void *)&arg1, (void *)&arg2); // Thread without lock
    thread_join();
    thread_join();
    thread_join();

    exit();
}
```


Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star8
init: starting sh
$ ./testthreads
below should be sequential print statements:
1. this should print first
1. sleep for 100 ticks
2. this should print second
2. sleep for 100 ticks
3. this should print third
3. sleep for 100 ticks
below should be a jarbled mess:
1. this 2. this should print whenever
2. sleep for 100 ticks
3. this should should print whenever
1. sleep for 100print whenever
ticks
3. sleep for 100 ticks
$

Scheduling:

برای پیاده‌سازی قسمت scheduling تابع scheduler را تغییر دادیم. این scheduler به تخصیص fair زمان CPU میان processها و threadهای مربوط به آنها می پردازد.

Process & Thread Examination:

در ابتدا scheduler روی هر process به ترتیب در process table پیمایش انجام می‌دهد. اگر process در حالت RUNNABLE نباشد و یا یک thread باشد، آن را skip می‌کند.

Checking for Process Children:

اگر یک process دارای فرزند باشد (که توسط childCount مشخص می‌شود)، scheduler زمان اجرا را به صورت fair بین process والد و فرزندانش تقسیم می‌کند. این کار با استفاده از متغیر turn انجام می‌شود که نوبت را مشخص می‌کند.

Executing Parent Process:

اگر نوبت process والد باشد، اجرا می‌شود.

Executing Child Processes:

اگر نوبت یکی از processهای فرزند باشد، scheduler روی تمامی فرزندان پیمایش می‌کند و فرزندی که نوبتش است را اجرا می‌کند.

Switching Turns:

بعد از اجرای یک process یا thread، مقدار متغیر turn آپدیت می‌شود تا مطمئن شویم که تمامی بچه‌ها و خود process به طور fair از زمان اجرا برخوردار می‌شوند.

Executing Processes Without Children:

اگر یک process فرزندی نداشته باشد، اجرا می‌شود.

در مجموع، این scheduler یک حلقه بی‌نهایت را اجرا می‌کند و به صورت مداوم process table را به منظور یافتن processهایی در حالت RUNNABLE پیمایش می‌کند تا آنها را اجرا کند.

```
void scheduler(void)
{
  struct proc *p , *childProc ;
  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 && !(p->state == SLEEPING  && p-
>childCount>0 )) || p->IsThread == 1 )
      {
        continue;
      }
      if ( p->childCount != 0 &&  p->turn == 0)
      {
        if(p->state == SLEEPING )
        {
          p->turn = 1 ;
        }
        else{
          // SwichProc(p);
          struct cpu *c = mycpu();
          c->proc=p;
          switchuvm(p);
          p->state = RUNNING;
          cprintf("process :%d \n" , p->pid );
          swtch(&(c->scheduler), p->context);
          switchkvm();
          c->proc = 0;
        }
        if ( p->turn != 0 )
        {
          int count = 1;
          for(childProc = ptable.proc; childProc <
&ptable.proc[NPROC]; childProc++)
          {
            if ( childProc->parent == p && childProc->IsThread==1 &&
 childProc->state == RUNNABLE ) // && childProc->state == RUNNABLE
            {
              // if ( childProc->state == RUNNABLE )
              // {
                if (count == p->turn)
                {
                  // SwichProc(childProc);
                  struct cpu *c = mycpu();
                  c->proc=childProc;
                  switchuvm(childProc);
                  childProc->state = RUNNING;
                  cprintf("process :%d \n" , childProc->pid );
                  swtch(&(c->scheduler), childProc->context);
                  switchkvm();
                  c->proc = 0;

                  break;
                }
                count++ ;
              // }
            }
          }
        }
        p->turn = (p->turn+1)%(p->childCount+1) ;
      }
      else
      {
        // SwichProc(p) ;
        struct cpu *c = mycpu();
        c->proc=p;
        switchuvm(p);
        p->state = RUNNING;
        cprintf("process :%d \n" , p->pid );
        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;
      }
    }
    release(&ptable.lock);
  }
}
```

لینک ریپازیتوری گیت هاب:

https://github.com/Computer-Engineering-Projects-IUST/OS-Final-Project