به نام خدا

گزارش پروژه پایانی سیستم عامل

غزل عربعلی (۹۷۵۲۱۳۹۶) – نیوشا یقینی (۹۸۵۲۲۳۴۶)

در ابتدا در فایل syscall.h دو سیستم کال افزوده شده با نام های زیر را اضافه میکنیم .

```
#define SYS_clone 22
#define SYS_join 23
```

سپس در فایل syscall.c از ۲ سیستم کال sys_clone و sys_join دو تابع با نام های زیر خروجی میگیریم .

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

همچنین باید این دو سیستم کال را به لیست سیستم کال های سیستم جهت استفاده اضافه کنیم .

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

در گام بعدی در فایل usys.S سیستم کال ها را به لیست سیستم کال هایی که باید با زبان اسمبلی که در سطح kernell کار میکنند اضافه میکنیم .

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

سپس درفایل defs.h تعریف توابع clone و join را اضافه میکنیم .

```
int  clone(void (*function)(void*), void*, void*);
int  join(int, void**);
```

سپس در فایل user.h این توابع را به توابع سطح کاربر اضافه میکنیم . (مانند عکس بالا)

سپس به تغییر فایل sysproc.c میپردازیم و توابع clone و join را تکمیلا تعریف میکنیم .

```
int sys_clone(void)
{
    int function, arg, stack;

    if(argint(0, &function) < 0)
        return -1;

    if(argint(1, &arg) < 0)
        return -1;

    if(argint(2, &stack) < 0)
        return -1;

    return clone((void *)function, (void *)arg, (void *)stack);
}
```

```
int sys_join(void)
{
    int tid, stack;

    if(argint(0, &tid) < 0)
        return -1;

    if(argint(1, &stack) < 0)
        return -1;

    return join(tid, (void **)stack);
}
```

حال به تغییر تابع wait در فایل proc.c میپردازیم : دلیل این تغییر آن است که هنگامی که ما از thread ها استفاده میکنیم pagetable ها کپی میشوند و در صورتی که ما د ر یک thread باشیم نباید این thread موجب توقف فرآیند والد شود.

```
if (p->parent != curproc&& p->pgdir != curproc->pgdir)
    continue;
```

تغییر بعدی در تابع scheduler اتفاق می افتد در این تابع با استفاده از الگوریتم RoundRobin بخاطر شبیه سازی thread ها با process ها جوری تابع را تغییر میدهیم که اولویت فرآیند ها در تخصیص منابع بیشتر از thread ها باشد

```
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 && p->thread_count==0)|| p-
>is_thread==1)
                  continue;
              //cprintf("this process selected pid: %d name: %s\n",p->pid,p->name);
              // Switch to chosen process.  It is the process's job
              // to release ptable.lock and then reacquire it
              // before jumping back to us.
              if(p->thread_count>0){
                  if(p->turn==0){
                      if(p->state!=RUNNABLE){
                          p->turn=1;
                          //cprintf("im here now");
                      }
                      c->proc = p;
                      switchuvm(p);
                      p->state = RUNNING;
                      //cprintf("pid :%d name: %s\n",p->pid,p->name);
                      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;
                  }if(p->turn!=0){
                      int index=0;
                      struct proc *p2;
                      for(p2 = ptable.proc; p2 < &ptable.proc[NPROC]; p2++){
                          if(p2->state != RUNNABLE|| p2->is_thread==0|| p2-
>parent!=p)
                              continue;
                          index++;
                          if(index==p->turn){
                              c->proc = p2;
                              switchuvm(p2);
                              p2->state = RUNNING;
                              //cprintf("pid :%d index of thread: %d name:
%s\n",p->pid,index,p->name);
                              swtch(&(c->scheduler), p2->context);
                              switchkvm();

                              // Process is done running for now.
                              // It should have changed its p->state before
coming back.
                              c->proc = 0;
                              break;
                          }
                      }
                  }
                  p->turn=(p->turn+1)%(p->thread_count+1);
              }
              else{
                  c->proc = p;
                  switchuvm(p);
                  p->state = RUNNING;
                  cprintf("pid :%d name: %s\n",p->pid,p->name);
                  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);

      }
}
```

سپس ۲ سیستم کال
clone و

```
int join(int tid, void **stack) {
    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 || p->pgdir != curproc->pgdir || p->pid !=
tid)
                continue;
            havekids = 1;
            if (p->state == ZOMBIE) {
                // Found one.
                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                *stack = p->tstack;
                p->tstack = 0;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                p->state = UNUSED;

                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
    }
}
```

وسیستم کال join را تعریف میکنیم

```
int clone(void (*function)(void *), void *arg, void *stack) {
    int i, pid;
    struct proc *np;
    struct proc *curproc = myproc();

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

    // <Code for new thread>

    // Copy process data to new process with page table address
    np->sz = curproc->sz;
    np->pgdir = curproc->pgdir;
    np->parent = curproc;
    *np->tf = *curproc->tf;

    // Stack pointer is at the bottom, bring it up, push return
    // address and arg
    *(uint *) (stack + PGSIZE - 1 * sizeof(void *)) = (uint) arg;
    *(uint *) (stack + PGSIZE - 2 * sizeof(void *)) = 0xFFFFFFFF;

    // Set esp (stack pointer register) and ebp (stack base register)
    // eip (instruction pointer register)
    np->tf->esp = (uint) stack + PGSIZE - 2 * sizeof(void *);
    np->tf->ebp = np->tf->esp;
    np->tf->eip = (uint) function;

    // Set thread stack
    np->tstack = stack;

    // </Code for new thread>

    // 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;
}
```

```c
int growproc(int n)
{
    uint sz;
    struct proc *curproc = myproc();
    struct proc *p;

    sz = curproc->sz;
    if(n > 0){
        if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
        {
            release(&ptable.lock);
            return -1;
        }
    } else if(n < 0){
        if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
        {
            release(&ptable.lock);
            return -1;
        }
    }
    curproc->sz = sz;

    // Change size of page table of all threads
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pgdir != curproc->pgdir)
            continue;

        p->sz = sz;
        switchuvm(p);
    }
    release(&ptable.lock);

    switchuvm(curproc);
    return 0;
}
```

سپس به تغییر تابع growproc میپردازیم چرا که در حالت عادی زمانی که یک فرایند داریم سایز page table مختص به آن تغییر میکند ولی زمانی که ما در این حالت thread داریم و رشد کرده و حافظه تخصیص یافته به آن بیشتر میشود باید آن را در فرایند والد نیز تاثیر دهیم چرا که page table های مشابه را استفاده میکنند .

حال تغییرات بعدی من جمله is_thread برای آن است که امکان اضافه شدن thread به thread دیگر نباشد و همچنین مشخص کردن نوبت thread با استفاده از int turn و int thread_count که تعداد thread های موجود در فرایند رو برای ما مشخص میکند که جهت schedule کردن فرایند ها از این موارد مطرح شده استفاده میشود.

```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 *tstack;                // thread stack
  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];
  int is_thread;// Process name (debugging)
  int turn;
  int thread_count;
}
```

تغییرات بعدی مربوط به ساخت ۲ فایل thread.h و thread.c هست .در فایل thread.c ۲ تابع thread_create و تابع thread_join پیاده سازی شده اند و درفایل thread.h نیز header های این توابع جهت استفاده از این توابع در جاهای دیگر قرار داده شده است .

```c
//thread.c file
#include "types.h"
#include "user.h"
#include "mmu.h"
#include "thread.h"
#include "x86.h"

int thread_create(void (*function) (void *), void *arg)
{
    void *stack = malloc(PGSIZE);

    if (stack == 0)
        return -1;
    if ((uint)stack % PGSIZE != 0)
        stack += PGSIZE - ((uint)stack % PGSIZE);

    return clone(function, arg, stack);
}

int thread_join(int tid)
{
    int retval;
    void *stack;
    retval = join(tid, &stack);
    free(stack);
    return retval;
}
```

```c
int thread_create(void (*function) (void *), void *arg);
int thread_join(int tid);
```

تغییرات بعدی مربوط به ساخت ۲ فایل lock.h و lock.c هست که در فایل تست از آنها استفاده شده.

```c
#include "types.h"
#include "user.h"
#include "mmu.h"
#include "lock.h"
#include "x86.h"
int lock_init(lock_t *lock)
{
    lock->flag = 0;
    return 0;
}

void lock_acquire(lock_t *lock)
{
    while(xchg(&lock->flag, 1) != 0);
}

void lock_release(lock_t *lock)
{
    xchg(&lock->flag, 0);
}
```

```c
typedef struct __lock_t{
    uint flag;
}lock_t;

int lock_init(lock_t *lk);
void lock_acquire(lock_t *lk);
void lock_release(lock_t *lk);
```

در نهایت از فایل روبرو جهت سنجش کارکردن همزمان thread ها استفاده میکنیم .

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "thread.h"
#include "lock.h"

#define ARRAY_SIZE 5
#define THREAD_NUMBER 3

int result[ARRAY_SIZE];
int numbers[ARRAY_SIZE];
lock_t *lock;


void f(void *arg) {
    int thread_number = *((int *)arg);

    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        lock_acquire(lock);
        int output = numbers[i] * (thread_number + 1);
        result[i] += output;
        printf(1, "Thread %d calculate element %d: %d  now
result is: ", thread_number + 1, i + 1, output);
        for (int i = 0; i < ARRAY_SIZE; i++)
        {
            printf(1, "%d ", result[i]);
        }
        printf(1, "\n");
        lock_release(lock);
    }

    exit();

}

int main() {
    //initilize sharing resource
    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        result[i] = 0;
        numbers[i] = i + 1;
    }

    int input_func[THREAD_NUMBER];
    int tid[THREAD_NUMBER];

    lock_init(lock);
    for (int i = 0; i < THREAD_NUMBER; i++) {
        input_func[i]=i;
        tid[i] = thread_create(&f, &input_func[i]);
    }


    for (int i = 0; i < THREAD_NUMBER; i++) {
        thread_join(tid[i]);
    }

    exit();
}
```