HW 4 Notes as creatively copying

Part 1

总的来说和HW3的完成方法类似，同时在作业网站中也给了很多提示去完成。

> Define three new system calls: first one to create a kernel thread, called `thread_create()`, second to wait for the thread to finish `thread_join()`, and then a third one that allows the thread to exit `thread_exit()`.

以上的这些system call的实现都是在`proc.c`中进行的

```
/*
 * This call creates a new kernel thread which shares the address space with the
 * calling process. In our implementation we will copy file descriptors in the same
 * manner fork() does it. The new process uses stack as its user stack, which is passed
 * the given argument arg and uses a fake return PC (0xffffffff). The stack should be
 * one page in size. The new thread starts executing at the address specified by fcn .
 * As with fork(), the PID of the new thread is returned to the parent.
 */
int thread_create(void(*fcn)(void*), void *arg, void*stack)
```

基本上这个是复制`proc.c`中的`fork()`函数的实现方式，但是需要注意的是在`thread_create`中

```
    np->pgdir = curproc->pgdir //new thread shares the same page table directory
```

page table的复制方式有别

根据提示，我们如此利用传进来的参数

```
    int i, pid, fakeret = FAKERET;
    //thread stack and return address mods
    stack += PAGESIZE - 8;
    memmove(stack + 4, &arg, sizeof(int));
    memmove(stack, &fakeret, sizeof(int));
    np->tf->esp = (int)stack; //new process uses stack as its user stack
    np->tf->eip = (int)fcn; //new thread starts executing at the address
specified by fcn
    //end of thread stack and return address mods
```

至此编辑完毕，我们继续到 `int thread_join(void)` ，根据作业所说

> This call waits for a child thread that shares the address space with the calling process. It returns the PID of waited-for child or -1 if none.

相较于wait，只需要删掉一行

```
        // removed from wait()
        // since thread still needs the same pgdir
        // freevm(p->pgdir);
```

作业提示中说

> Finally, the `int thread_exit(void)` system call allows a thread to terminate.

但其实在 `proc.c` 中的 `exit` 并不是一个int返回值的函数，应该是老师的笔误，其实是一个全部复制exit()但是返回类型是void的函数，如下

```
void thread_exit(void);
```

根据之前HW3我们学到的更改方法

`user.h`

```
int thread_create(void(*)(void*), void*, void*);
int thread_join(void);
// void thread_exit(void);
```

solved ~~*不知道签名正确与否，* `thread_create` *参数形式，以及* `thread_exit` *的返回类型*~~

观察 `exit` 的添加方式

```
int exit(void) __attribute__((noreturn));
```

但其实在 `proc.c` 里写的是void，所以我们也效仿

```
int thread_exit(void) __attribute__((noreturn));
```

`user.S`

```
SYSCALL(thread_create)
SYSCALL(thread_join)
SYSCALL(thread_exit)
```

`defs.h`

```
int             thread_create(void(*fcn)(void*), void *arg, void*stack);
int             thread_join(void);
void            thread_exit(void);
```

`syscall.c`

```
extern int sys_thread_create(void);
extern int sys_thread_join(void);
extern int sys_thread_exit(void);
...
[SYS_thread_create]  sys_thread_create,
[SYS_thread_join]    sys_thread_join,
[SYS_thread_exit]    sys_thread_exit,
```

我们需要从`sysproc.c`中调用system call，然后去调用我们写好的函数

`sysproc.c`

```
int
sys_thread_create(void)
{
  char* fcn;
  char* arg;
  char* stack;

  if (argptr(0, &fcn, 4) < 0)
    cprintf("%s\n", "system call argument pointer fct fetch error");
  if (argptr(1, &arg, 4) < 0)
    cprintf("%s\n", "system call argument pointer arg fetch  error");
  if (argptr(2, &stack, 4) < 0)
    cprintf("%s\n", "system call argument pointer stack fetch  error");

  //return thread_create(void(*fcn)(void*), void *arg, void*stack);
        return thread_create((void (*)(void *))fcn, arg, stack);
```

```
}

int
sys_thread_exit(void)
{
  thread_exit();
  return 0;   // not reached
}

int
sys_thread_join(void)
{
  return thread_join();
}
```

return的调用不知道为什么要那个样子，不是很清楚

因为我们需要使用一个给出的`thread.c`文件来进行结果的验证，所以在

`Makefile`

```
      _thread\
```

编译后我们调用thread

```
$ thread
Starting do_work: s:b1
Starting do_work: s:b2
Done s:2FAC
Done s:2F88
Threads finished: (4):5, (5):4, shared balance:3200
```

我们会发现`shared balance`在一个不是为6000的值，这是因为我们没有弄spinlock等part 2要求的内容

Part 2

> If you implemented your threads correctly and ran them a couple of times you might notice that the total balance (the final value of the `total_balance` does not match the expected 6000, i.e., the sum of individual balances of each thread. This is because it might happen that both threads read an old value of the `total_balance` at the same time, and then update it at almost the same time as well. As a result the deposit (the increment of the balance) from one of the threads is lost.

Spinlock

> To fix this synchronization error you have to implement a spinlock that will allow you to execute the update atomically, i.e., you will have to implement the `thread_spin_lock()` and `thread_spin_unlock()` functions and put them around your atomic section (you can uncomment existing lines above).
>
> Specifically you should define a simple lock data structure and implement three functions that: 1) initialize the lock to the correct initial state (`void thread_spin_init(struct thread_spinlock *lk)`), 2) a funtion to acquire a lock (`void thread_spin_lock(struct thread_spinlock *lk)`), and 3) a function to release it `void thread_spin_unlock(struct thread_spinlock *lk)` .
>
> To implement spinlocks you can copy the implementation from the xv6 kernel. Just copy them into your program (`threads.c` and make sure you understand how the code works).

去`spinlock.c`和`spinlock.h`中找到可以复制的内容

虽然不知道干了什么但是复制了一下就可以用了，删掉了一些可能可以作为调试用信息的代码，去掉了头文件中`defs.h`，就可以用了

`thread.c`

```c
#include "param.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"


// struct for thread spin lock
struct thread_spinlock {
    uint locked;       // Is the lock held?

    // For debugging:
    char *name;        // Name of lock.
    struct cpu *cpu;   // The cpu holding the lock.
    uint pcs[10];      // The call stack (an array of program counters)
    // that locked the lock.
};

void thread_spin_init(struct thread_spinlock *lk, char *name){
    lk->name = name;
    lk->locked = 0;
    lk->cpu = 0;
}
```

```c
void thread_spin_lock(struct thread_spinlock *lk){
    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();
}

void thread_spin_unlock(struct thread_spinlock *lk){

    lk->pcs[0] = 0;
    lk->cpu = 0;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that all the stores in the critical
    // section are visible to other cores before the lock is released.
    // Both the C compiler and the hardware may re-order loads and
    // stores; __sync_synchronize() tells them both not to.
    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    lk->locked = 0;
}
```

这个时候我们可以看到输出变成了6000

```
$ thread
Starting do_work: s:b1
Starting do_work: s:b2
Done s:2FAC
Done s:2F88
Threads finished: (4):5, (5):4, shared balance:6000
```

mutex与spinlock基本一致，不过将忙等变为让行，但是xv6中没有yield，所以用sleep代替

```c
// struct for thread mutex lock
struct thread_mutex {
    uint locked;       // Is the lock held?

    // For debugging:
    char *name;        // Name of lock.
```

```
    struct cpu *cpu;    // The cpu holding the lock.
    uint pcs[10];       // The call stack (an array of program counters)
    // that locked the lock.
};

void thread_mutex_init(struct thread_mutex* lk, char* name){
    lk->name = name;
    lk->locked = 0;
    lk->cpu = 0;
}

void thread_mutex_lock(struct thread_mutex *lk){
    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        sleep(1);

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();
}

void thread_mutex_unlock(struct thread_mutex *lk){
    lk->pcs[0] = 0;
    lk->cpu = 0;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that all the stores in the critical
    // section are visible to other cores before the lock is released.
    // Both the C compiler and the hardware may re-order loads and
    // stores; __sync_synchronize() tells them both not to.
    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    lk->locked = 0;
}

struct thread_spinlock lock;
struct thread_spinlock* lk = &lock;
struct thread_mutex mutex;
struct thread_mutex* m = &mutex;
char name[20] = "thread_spin_lock";
```

注意调用的时候传参形式

```
thread_mutex_lock(m);
...
thread_mutex_unlock(m);
```

此时我们可以看到发生了一些细微改变

```
$ thread
Starting do_work: s:b1
Starting do_work: s:b2
Done s:2F88
Done s:2FAC
Threads finished: (4):4, (5):5, shared balance:6000
```

基本作业做完