CS344 : OS Lab, Assignment-1
Group-C33

Name: Mukul Garg Roll no. : 200101068
Name: Arpan Khanna Roll no. : 200101121
Name: Kartikey Gupta Roll no. : 200101055

PART A:

We defined three system call functions for thread creation, joining and exiting. Their corresponding calling functions are defined in **proc.c**

```c
int thread_create(void (*fcn)(void*), void *arg, void *stack)
{
  // how to pass argument in
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

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

  np->pgdir = curproc->pgdir;

  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Set eip instruction poitner
  // Set esp stack pointer
  // fcn is address of function ?
  np->tf->eax = 0;
  np->tf->eip = (uint)fcn;
  np->tf->esp = (uint)stack+4096;

  // put arg into user stack
  // set the context of np->tf->esp to the address of arg
  np->tf->esp -= 4;
  *(uint*)(np->tf->esp) = (uint)(arg);

  np->tf->esp -= 4;
  *(uint*)(np->tf->esp) = (uint)0xFFFFFFF;


  // 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 thread_join(void)
{
  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;
        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 toexit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

```c
int thread_exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

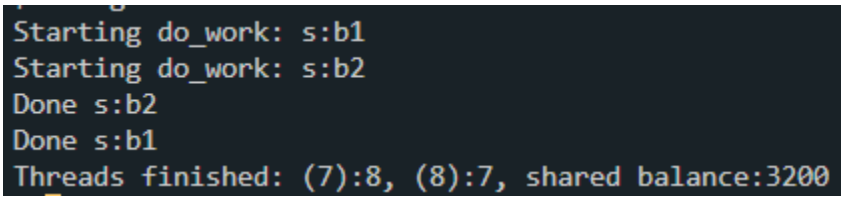
thread_exit function to terminate the thread

```c
#include "types.h"
#include "stat.h"
#include "user.h"
struct balance
{
 char name[32];
 int amount;
};
volatile int total_balance = 0;
volatile unsigned int delay (unsigned int d)
{
 unsigned int i;
 for (i = 0; i < d; i++)
 {
 __asm volatile( "nop" ::: );
 }
 return i;
}
void do_work(void *arg)
{
 int i;
 int old;
 struct balance *b = (struct balance*) arg;
 printf(1, "Starting do_work: s:%s\n", b->name);
 for (i = 0; i < b->amount; i++)
 {
 old = total_balance;
 delay(10000);
 total_balance = old + 1;
 }
 printf(1, "Done s:%x\n", b->name);
 thread_exit();
 return;
}
int main(int argc, char *argv[]) {
 struct balance b1 = {"b1", 3200};
 struct balance b2 = {"b2", 2800};
 void *s1, *s2;
 int t1, t2, r1, r2;
 s1 = malloc(4096);
 s2 = malloc(4096);
 t1 = thread_create(do_work, (void*)&b1, s1);
 t2 = thread_create(do_work, (void*)&b2, s2);
 r1 = thread_join();
 r2 = thread_join();
 printf(1, "Threads finished: (%d):%d, (%d):%d, shared balance:%d\n",
 t1, r1, t2, r2, total_balance);
 exit();
```

Created a file **thread.c** with given function calls to create a thread then join the thread and then execute the same.

thread_create call creates a new kernel thread which shares the address space with the calling process.
The other new system call is thread_join. This call waits for a child thread that shares the address space with the calling process. It returns the PID of the waited-for child or -1 if none. Finally, the thread_exit system call allows a thread to terminate.

The thread_create() call should behave very much like fork(), except that instead of copying the address space to a new page directory, clone initializes the new process so that the new process and cloned process use the same page directory. Thus, memory will be shared, and the two "processes" are really actually threads.
The int thread_join(void) system call is very similar to the already existing int wait(void) system call in xv6. Join waits for a thread child to finish, and wait waits for a process child to finish. Finally, the thread_exit() system call is very similar to exit().



Output after running **thread.c**

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 both threads might read an old value of the total_balance at the same time and then update it at almost the same time. As a result, the deposit (the increment of balance) from one of the threads is lost.

PART B

Added a spinlock in thread.c to correct synchronization across threads, implemented three functions **thread_initlock, thread_spin_lock, thread_spin_unlock** that initialize the lock to the correct initial state

```
struct thread_spinlock {
  uint locked;       // Is the lock held?

  // For debugging
  char *name;        // Name of lock.
};

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



void thread_spin_lock(struct thread_spinlock* lk) {

    // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;
  __sync_synchronize();
}

void thread_spin_unlock(struct thread_spinlock* lk) {

  __sync_synchronize();

  asm volatile("movl $0, %0" : "+m" (lk->locked) : );
}
```

```
  for (i = 0; i < b->amount; i++) {
      thread_spin_lock(&lock);
      //mutex_lock(&m_lock);
       old = total_balance;
       delay(100000);
       total_balance = old + 1;
      thread_spin_unlock(&lock);
      // mutex_unlock(&m_lock);
  }
```

Using **thread_spin_lock()** and **thread_spin_unlock()** in the function

```
Starting do_work: s:b2
Starting do_work: s:b1
Done s:b2
Done s:b1
Threads finished: (4):5, (5):4, shared balance:6000
```

Output after using **spinlock**

Now, the final value of the **total_balance will match with the expected value 6000**, i.e., the sum of individual balances of each thread.

```
struct mutex_lock{
  uint locked;
};

void mutex_initlock(struct mutex_lock* lk) {
   lk->locked = 0;
}

void mutex_lock(struct mutex_lock* lk) {
  while(xchg(&lk->locked, 1) != 0)
    sleep(1);
  __sync_synchronize();
}

void mutex_unlock(struct mutex_lock* lk) {
  __sync_synchronize();

  asm volatile("movl $0, %0" : "+m" (lk->locked) : );
}
```

Implemented **mutex_lock** in **thread.c** which instead of spinning on a thread release the CPU to another thread

```
  for (i = 0; i < b->amount; i++) {
       //thread_spin_lock(&lock);
      mutex_lock(&m_lock);
       old = total_balance;
       delay(100000);
       total_balance = old + 1;
     //  thread_spin_unlock(&lock);
       mutex_unlock(&m_lock);
  }
```

Using **mutex_lock()** and **mutex_unlock()** in the function

```
Starting do_work: s:b1
Starting do_work: s:b2
Done s:b1
Done s:b2
Threads finished: (7):7, (8):8, shared balance:6000
```

Output after using **mutex_lock**

While spinlocks that we've implemented above implement correct synchronization across threads, they might be inefficient in some cases. For example, when all threads of the process run in parallel on different CPUs, spinlocks are perfect---each process enters a short critical section, updates the shared balance atomically and then releases the spinlock for other threads to make progress. However, if we are running on a system with a single physical CPU then it will become highly inefficient as all will wait for the interrupted (lock-holding) thread to be rescheduled again.

The following files were changed to implement the assignment:

Makefile

The makefile had to be edited to add the new user programs to test the creation of threads and the concurrent execution of code.

defs.h

The declarations for thread_create, thread_join and thread_exit were created in this file.

proc.c

The code definitions for thread_create, thread_join and thread_exit were added to this file. The thread_create function sets up a new process with the given stack arguments, and the thread_join function scans the process table looking for a zombie child and clears them out.

proc.h

A new property was added to the process data structure to mark the address of the thread stack, titled *ustack.

syscall.c

The declarations for the new functions thread_create, thread_join and thread_exit were added to this file.

syscall.h

This system call numbers were assigned to the new functions thread_create, thread_join and thread_exit.

sysproc.c

This file contains the definitions of the thread_create, thread_join and thread_exit functions which call the definitions in proc.c.

user.h

The declarations for the new system calls thread_create, thread_join and thread_exit were added to this file.

usys.S

The declarations for the new functions thread_create, thread_join and thread_exit were added to this file.

thread.c

This user program tests the creation of two different threads and their usage of locks and mutexes to ensure concurrency is working and thread safety is achieved via locks.
