Group no C38
Vatsal Gupta 200101105
Sweeya Reddy 200101079
Pranshu Kandoi 200101086

Assignment-1 OS344

Part 1: Kernel threads

About threads and target

Threads are very much like processes (they can run in parallel on different physical CPUs), with the difference that they **share the same address space** (the address space of the process that created them). Hence all threads of the same process can read and update all the variables in that address space to communicate and collaborate on computing a complex result in parallel.

In this part of the assignment, we need to implement these threads. In particular, we have to make 3 functions which aid us in their implementation -

- **thread_create()**
- **thread_join()**
- **thread_exit()**

<u>We followed the given Hints:</u>

- [ ] 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(). You should however be careful and do not deallocate the page table of the entire process when one of the threads exits.

Creating a thread- proc.c

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 that 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)
{
   int i, pid;
   struct proc *nt;
   struct proc *curproc = myproc();
   //Allocate thread.
   if((nt = allocproc()) == 0){
     return -1;
   }

   //just make up a flag saying that this entry is a thread.
   nt->thread = 1;
   //pagedir is shared hence unlike fork, we do not copy it.
   nt->pgdir = curproc->pgdir;
   //setting size and trapframe of new thread from parent process.
   nt->sz = curproc->sz;
   nt->parent = curproc;
   *nt->tf = *curproc->tf;
   uint sp; //stack pointer
   //creating a user stack.
   uint ustack[2];
   //making sure stack is one page in size
   sp = (uint)stack + PGSIZE;
   memset((char*)stack, 0, PGSIZE); //main may recycle stack page, so need to clear it first
   // tls is a small structure holding thread information like thread id and index, held at the top of
every stack created.
   struct tls t;
   t.tid = nt->pid - curproc->pid - 1; //tid starts from 0
   sp -= sizeof(struct tls);
   memmove((char*)sp, (char*)&t, sizeof(struct tls));
   //passing fake return PC to user stack
   ustack[0] = 0xffffffff;
   //passing arg to user stack
   ustack[1] = (uint)arg;
   //void do_work(void *arg)
   sp -= 2*4;
   if(copyout(nt->pgdir, sp, ustack, 2*4) < 0) return -1;
   //initialize thread's stack pointer
   nt->tf->esp = sp;
   // Clear %eax so that fork returns 0 in the child.
   //np->tf->eax = 0;
   //setting the instruction pointer to fcn from where the execution begins
   nt->tf->eip = (uint)fcn;
   //we will copy file descriptors in the same manner fork() does it.
   for(i = 0; i < NOFILE; i++)
     if(curproc->ofile[i])
       nt->ofile[i] = filedup(curproc->ofile[i]);
   //setting new thread's current working dfirectory to the same directory as the parent process.
   nt->cwd = idup(curproc->cwd);
   safestrcpy(nt->name, curproc->name, sizeof(nt->name));
   pid = nt->pid;
```

```
    acquire(&ptable.lock);
    nt->state = RUNNABLE;
    release(&ptable.lock);
    //As with fork(), the PID of the new thread is returned to the parent.
    return pid;
}
```

- Firstly, we allocate a proc pointer to point to the thread which we create as a new process. To this newly created thread, we allocate the shared page directory and set the same size and trapframe content as the parent thread. We also denote **nt-> thread = 1** to denote that this newly created process thread.
- Next, we intialise a user stack according to the specifications as in question. We pass the starting pointer of the location from where we want to store the stack as ***stack.** We copy the arg and the fake return PC to the user stack.
- eip register of the thread is set to intimate from where we want the execution to begin. This is passed as *fcn which points to the function which we want to execute.
  Esp register is set to the stack pointer which we created.
- Current working directory and openfiles and name of the parent process is copied for the thread.
- Lastly, lock is acquired for the process table and its state is changed to RUNNABLE and the lock is then released. And then the pid of the newly created thread is returned.

Joining a thread-proc.c

Most of the code is similar to wait() we just have to ensure that we don't free up the parent process page directory as unlike with fork and different processes, here the address space and the page directory is shared for different threads.

```
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 || p->thread != 1)
          continue;
        havekids = 1;
        if(p->state == ZOMBIE){
          // Found one.
          pid = p->pid;
          kfree(p->kstack);
          p->kstack = 0;
```

```
        p->pgdir = 0;//slight change here, explained below.
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        p->thread = 0;
        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
  }
  return -1;
}
```

- Firstly, we create two variables havekids and pid. These denote whether the calling process has any kids and what child is killed/ being waited for.
- Next, lock on the process table is acquired and checking if the current running process has any children.

1. And if so, if any of those children have reached a “ZOMBIE” or exit status then its corresponding variables are freed up, we clear the kernel stack, the pgdir pointer (making sure that unlike wait this isn’t freed up), also pid,parent,name and killed status is set to 0. The status is set to unused and its thread status is removed. The pid of that child is returned and the lock on process table is released.

2. If a child exists but hasn’t reached a “ZOMBIE” state, we make the current process go to sleep and wait for a child to reach a “ZOMBIE” state and exit.

- If the process doesn’t have any children(or if the current process has been killed), there is no point in waiting and the lock is released.

Exiting a thread-proc.c

Most of the code is similar to exit(). The main difference is only that while the existing with the exit() function works on processes we have to understand its functioning with respect to threads. The code and the explanation is as below:

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

```c
  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");
}
```

- Firstly, we confirm whether the current process that we are trying to exit is not the initproc, if so we enter a panic state where we intimate that the initproc is exiting!
- All open files of the curproc are closed
- The current working directory of the curproc is cached for future use.
- Lock is acquired on process table and the parent process is woken up.
- Next, we check whether the process we are exiting has any children which may be orphaned. If so, their parent is set to initproc and if any of these children already has a “ZOMBIE” state then initproc is woken up to ensure subsequent joining of these children.
- Lastly,curproc is exited and its state is set to ZOMBIE.

System Calls implementation

1. Firstly, changes are made to the **defs.h** file to declare the function prototypes which are defined in proc.c

```c
//PAGEBREAK: 16
// proc.c
int             cpuid(void);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
//thread links
int             thread_create(void(*fcn)(void*), void* arg, void* stack);
int             thread_join(void);
void            thread_exit(void);
// swtch.S
void            swtch(struct context**, struct context*);
```

2. **We will now create system calls** and make their corresponding links. Indexes for the system calls that we will implement are defined in **syscall.h**

```c
// syscall.h > SYS_link
// 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_draw 22
#define SYS_thread_create 23
#define SYS_thread_join   24
#define SYS_thread_exit   25
```

3. **usys.S** links the index number to the system call function.

```
usys.S
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(draw)

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

```

4. The function pointers to the index is done in **syscall.c**

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_draw]  sys_draw,
[SYS_thread_create] sys_thread_create,
[SYS_thread_join]   sys_thread_join,
[SYS_thread_exit]   sys_thread_exit,
};
```

5. These system calls are correspondingly defined in **sysproc.c**

```c
//
int
sys_thread_create(void) {
  int fcn;
  char* arg;
  char* stack;
  if (argint(0, &fcn) < 0)
    return -1;
  if (argstr(1, &arg) < 0)
    return -1;
  if (argstr(2, &stack) < 0)
    return -1;
  // fcn make integer to void(*)(void*)
  return thread_create((void(*)(void*))fcn, arg, stack);
}
int
sys_thread_exit(void)
{
  thread_exit();
  return 0;
}
int
sys_thread_join(void)
{
  return thread_join();
}
```

6. Sysproc.c is using kernel functions which are defined in proc.c as before and correspondingly declared in defs.h as explained above in point 1.
7. These thread functions are declared in **user.h** which are now available through system calls to the user. Following 3 lines are added:

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

Kind Note: **proc.h** is also changed to include the thread variable which intimates the identification of a thread

```c
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  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 thread;
};
```

Linking running of thread.c (given) and corresponding system calls

1. **thread** is listed under Makefile.

   Under UPROGS and EXTRA we add thread as shown below.

```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_drawtest\
	_thread
```

```
EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c drawtest.c thread.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

2. **thread.c** is as below:

C old_thread.c > delay(unsigned int)

```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++)
    {
        // thread_spin_lock(&lock);
        old = total_balance;
        delay(100000);
        total_balance = old + 1;
        // thread_spin_unlock(&lock);
    }
    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();
}
```

3. Since locks are not implemented, the shared balance isn't 6000 as intended and on each iteration gives a different unexpected value (around 3200)
4. <u>Please note:</u> Since thread.c is modified in later part of the assignment, this unsynchronized previous version of the file is also included in the code as old_thread.c and correspondingly linked following the same logic as thread.c as above.

5. The code is run after running **make clean** and **make-qemu-nox** in the terminal

```
$ old_thread
SSttaarrttiinngg  ddoo__wwoorkr:k :s :s:bb21

Done s:2F9C
Done s:2F78
Threads finished: (18):19, (19):18, shared balance:3201
$ old_thread
SSttaarrttiingng  ddoo_w_owrokr: ks:: sb:b2
1
Done s:2F9C
Done s:2F78
Threads finished: (21):22, (22):21, shared balance:3204
$ old_thread
SSttaarrttiingng  ddoo__wowrokr:k:  ss::bb21

Done s:2F9C
Done s:2F78
Threads finished: (24):25, (25):24, shared balance:3200
$
```

Part 2: Synchronization

Structure and function definitions

In thread.c, the below changes are made

1. Structures are made as below, following spinlock.h file

```
struct thread_spinlock lock;
struct mutex_lock m_lock;
```

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

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

struct mutex_lock
{
  char* name;
  uint locked;
};
```

2. Following spinlock.c file, initlock for spinlock and mutex is implemented as below to give starting values(name and free state to the locks)

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

3. Acquisition or lock_acquire function is implemented as below following spinlock.c file

```c
void thread_spin_lock(struct thread_spinlock *lk)
{

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

4. Unlock or release function is implemented as below again drawing inspiration from spinlock.c file

```c
void thread_spin_unlock(struct thread_spinlock *lk)
{

  __sync_synchronize();

  asm volatile("movl $0, %0"
               : "+m"(lk->locked)
               :);
}
void mutex_unlock(struct mutex_lock *lk)
{
  __sync_synchronize();

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

5. Lastly, do_work function is changed to lock during its critical section using either spinlock or mutex as implemented above

```c
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++)
  {
    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);
  }

  printf(1, "Done s:%s\n", b->name);

  thread_exit();
  return;
}
```

6. Running thread now, we see the balance achieved to be 6000 as the process are implemented correctly concurrently and in a synchronized way.
   (same output using spinlocks/mutex as below)

```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ thread
SSttaarrttiinngg  ddoo__wwoork: s:b2
rk: s:b1
Done s:b2
Done s:b1
Threads finished: (4):5, (5):4, shared balance:6000
$ thread
SSttaarrttiinngg  ddoo__wwoorrkk::  s:s:b2b
1
Done s:b2
Done s:b1
Threads finished: (7):8, (8):7, shared balance:6000
$ thread
SSttaarrttiinngg  ddoo__wwoorrkk::  ss::bb12

Done s:b2
Done s:b1
Threads finished: (10):11, (11):10, shared balance:6000
$
```

Part 3: Patch file creation

1. We copied the original xv6-public folder again to create a patch file between the newly edited (copy-xv6) and original (xv6-public) folder.
2. We then made a new folder to try out the patch we had constructed
3. We tested the patch and re-ran thread to ensure that it was working correctly.

```
vatsal@vatsal-lab:~/os-assignment1$ diff -ruN xv6-public/ copy-xv6/ > file.patch
vatsal@vatsal-lab:~/os-assignment1$ mkdir tryingOutPatch
vatsal@vatsal-lab:~/os-assignment1$ cd tryingOutPatch/
vatsal@vatsal-lab:~/os-assignment1/tryingOutPatch$ # patch file and xv6-public copied to tryingOutPatch
vatsal@vatsal-lab:~/os-assignment1/tryingOutPatch$ patch -s -p0 < file.patch
```

4. Make-clean and then make-qemu-nox was run on the xv-6 public folder
5. Then the following output was witnessed on running thread.

```
$ thread
SSttaarrttiinngg  ddoo__wwoorrkk::  ss::bb12

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