CS344 Assignment-1

Name: Dhruv Tarsadiya Roll No: 200101123
Name: Vikram Singla Roll No: 200101117
Name: Harshwardhan Bhakkad Roll No: 200101040

---

1 Kernel Threads

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. Threads in the same process share:

- Process instructions
- Most data
- Open files (descriptors)
- Signals and Signal handlers
- Current Working Directory

Each thread has unique:

- Thread ID
- Set of registers, Stack pointer
- Stack for local variables, Return addresses
- Priority

We implement the three system calls **thread_create()**, **thread_join()**, and **thread_exit()** using the function descriptions of the similar process counterparts **fork()**, **wait()**, and **exit()**. Let us take a look at these functions one by one.

1.1 thread_create()

The **thread_create()** call behaves very much like **fork()**, except that instead of copying the address space to a new page directory, the child process initializes the new process so that the new process and the cloned process use the same page directory. Thus, memory will be shared, and the two "processes" are really actually threads.

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

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

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // 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;
}
```

(a) System Call : fork()

```
int thread_create(void (*fcn)(void *), void * arg, void * stack) {
  if ((uint) stack == 0) {
    return -1;
  }
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();
  // Allocate process.
  if ((np = allocproc()) == 0)    return -1;
  // Threads share the same page table as the parent
  np->pgdir = curproc->pgdir;
  np->parent = curproc;
  np->sz = curproc->sz;
  *np->tf = *curproc->tf;

  // Mark this proc as a thread
  np->isThread = 1;

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

  // set function
  np->tf->eip = (int) fcn;

  // We set stack pointer to the end of the stack
  np->tf->esp = (int) stack + 4096;
  np->tf->esp -= 4;
  // Store arguments of the fcn in the stack
  *((int*) (np->tf->esp)) = (int) arg;
  np->tf->esp -= 4;
  // Fake PC Value
  *((int*) (np->tf->esp)) = 0xffffffff;

  // filedup gives the file descriptors and increments them by one inorder
  //to keep track of the number of instances of a file used by proccesses;
  // If no instance is being used then the OS may close the file
  for (i = 0; i < NOFILE; i++)
    if (curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);
  // Process name and Process ID
  safestrcpy(np->name, curproc->name, sizeof(curproc->name));
  pid = np->pid;
  // Thread is ready to run
  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);
  return pid;
}
```

(b) System Call : thread_create()

While calling fork() [Figure 1] we copy process state from parent; on the other hand the new spawned thread must share the same page table as the parent. The Page Table holds the mapping of the virtual addresses to the physical addresses and hence, threads must share that with the parent since they share address space. The **Extension Instruction Pointer (EIP)** will hold the function; it basically tells the computer where to go next to execute the next command. We set stack pointer**(ESP)** to the end of the stack and store arguments of the fcn in the stack; we copy the file descriptors from the parent process and change the new thread's state to "**RUNNABLE**" which is equivalent to Ready State.

1.2 thread_join()

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. The **int thread_join(void)** system call

```c
int
wait(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 to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

(a) System Call : wait()

```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->isThread == 0 || 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 to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

(b) System Call : thread_join()

is very similar to the already existing **int wait(void)** system call in xv6. We only need to take care not to deallocate the page table. Join waits for the thread child to finish.

1.3 thread_exit()

The **int thread_exit(void)** system call allows a thread to terminate. We have taken care earlier that the page table of the entire process is not deallocated when one of the thread is terminated. The relevant pictures are included below.

We made the following changes in these files:

1. Implemented system calls in **proc.c**
2. Added a pointer to the system call in this **syscall.h**. This file contains an array of function pointers which use indices as pointers to system calls. We will update it to include our custom call.
3. Added external functions visible to the whole program in **syscall.c** . It connects the shell and the kernel, and the system call function which we added to the system call vector in **syscall.h**
4. Added prototypes to system calls in **syproc.c** and **defs.h** and **user.h**
5. **usys.S** : Creates a user level system call definition for our system calls.

```c
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;
  if(curproc == initproc)
    panic("init exiting");
  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);
  wakeup1(curproc->parent);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

(a) System Call : exit()

```c
int thread_exit() {
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;
  if (curproc == initproc)
    panic("init exiting");
  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);
  wakeup1(curproc->parent);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->parent == curproc) {
      p->parent = initproc;
      if (p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

(b) System Call : thread_exit()

1.4 thread.c

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 (3200 + 2800). 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. This can be seen in the image(a) below.

```
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 b
t 58
init: starting sh
$ thread
SSttararting do_worrtik:ng s :dbo_1wor
k: s:b2
Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:3204
```

(a) Output on VGA on executing thread.c using 3200 and 2800 balances

```
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ thread
SSttaarrtiting do_work: s:b2
ng do_work: s:b1
Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:4521
$
```

(b) Output on VGA on executing thread.c using 4500 and 4500 balances

Even in our second test case where the individual balances were 4500 each, the total balance never exceeded 9000. This can be seen by the image(b) above.

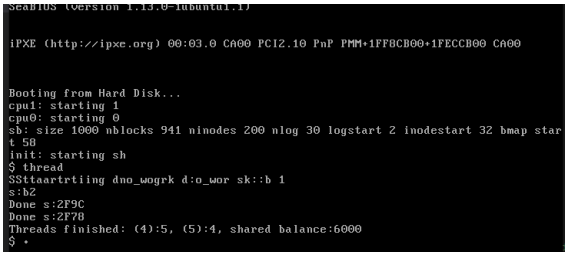
2 Thread Synchronization

We made **spinlocks** by implementing three functions :

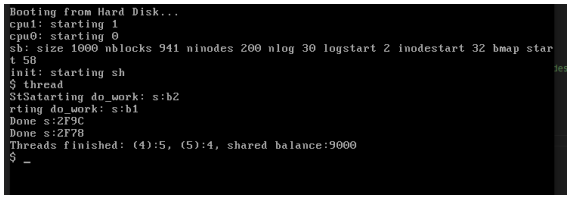
1. Initialize lock to correct state **(void thread_spin_init(struct thread_spinlock *lk))**
2. A funtion to acquire a lock **(void thread_spin_lock(struct thread_spinlock *lk))**
3. A function to release it **(void thread_spin_unlock(struct thread_spinlock *lk))**

These functions are implemented in the file **lockFunc.h**.

This spinlock can be compared to *compare_and_swap* atomic instruction. It uses the *xchg* assembly instruction to swap two values and return the original value of the variable. Hence, once a thread acquires a lock; the other threads get stuck in while loop.



(a) Output on VGA on executing thread.c using 3200 and 2800 balances with thread synchronization. We get 6000 as output.



(b) Output on VGA on executing thread.c using 4500 and 4500 balances with thread synchronization. We get 9000 as output.

Similarly for **mutexes** we have implemented three functions:

1. Initialize mutex to correct state **(void thread_mutex_init(struct thread_mutex *m))**
2. A function to acquire a mutex **(void thread_mutex_lock(struct thread_mutex *m))**
3. A function to release it **(void thread_mutex_unlock(struct thread_mutex *m))**

The main difference between spinlock and mutex is that spinlock is a lock which causes a thread trying to acquire it to simply wait in the loop and repeatedly check for its availability. In contrast, a mutex will allow you to execute the update atomically similar to spinlock, but instead of spinning it will release the CPU to another thread. To implement mutex we have used the sleep() system call. This difference can be seen using the images below.

```
void thread_mutex_lock(struct thread_mutex *m) {
    while (xchg(&m->lock, 1) != 0)
        sleep(1);
    __sync_synchronize();
};
```

(a) Acquiring mutex lock

```
void thread_spin_lock(struct thread_spinlock *lk) {
    while (xchg(&lk->lock, 1) != 0)
        ;
    __sync_synchronize();

};
```

(b) Acquiring spin lock