CS_344 Assignment 1

Group C21: Satvik Tiwari, 200101091

Pranjal Baranwal, 200101083

Akshat Mittal, 200101011

Task: Kernel Threads and Synchronization

Part 1: Kernel Threads

In this part, we have to implement kernel threads using three system calls:

1. ***thread_create()***

   This will be used to create a new kernel thread which shares same address space. Returns PID of new thread to parent.

   Function definition:

   int thread_create(void(*fcn)(void*), void *arg, void*stack)

```
180 int
181 fork(void)
182 {
183   int i, pid;
184   struct proc *np;
185   struct proc *curproc = myproc();
186 
187   // Allocate process.
188   if((np = allocproc()) == 0){
189     return -1;
190   }
191 
192   // Copy process state from proc.
193   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
194     kfree(np->kstack);
195     np->kstack = 0;
196     np->state = UNUSED;
197     return -1;
198   }
199   np->sz = curproc->sz;
200   np->parent = curproc;
201   *np->tf = *curproc->tf;
202 
203   // Clear %eax so that fork returns 0 in the child.
204   np->tf->eax = 0;
205 
206   for(i = 0; i < NOFILE; i++)
207     if(curproc->ofile[i])
208       np->ofile[i] = filedup(curproc->ofile[i]);
209   np->cwd = idup(curproc->cwd);
210 
211   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
212   pid = np->pid;
213   acquire(&ptable.lock);
214   np->state = RUNNABLE;
215   release(&ptable.lock);
216   return pid;
217 }
```

Fig. *fork()*

```
527 int
528 thread_create(void (*fcn)(void *), void * arg, void * stack)
529 {
530   if ((uint) stack == 0) {
531     return -1;
532   }
533   int i, pid;
534   struct proc *np;
535   struct proc *curproc = myproc();
536   // Allocate process.
537   if ((np = allocproc()) == 0)
538     return -1;
539   np->pgdir = curproc->pgdir;
540   np->sz = curproc->sz;
541   np->parent = curproc;
542   *np->tf = *curproc->tf;
543   // Mark this proc as a thread
544   np->is_thread = 1;
545   // Clear %eax so that fork returns 0 in the child.
546   np->tf->eax = 0;
547   // set function
548   np->tf->eip = (int) fcn;
549   np->tf->esp = (int) stack + 4096;
550   np->tf->esp -= 4;
551   *((int*) (np->tf->esp)) = (int) arg;
552   np->tf->esp -= 4;
553   *((int*) (np->tf->esp)) = 0xffffffff;
554 
555   for (i = 0; i < NOFILE; i++)
556     if (curproc->ofile[i])
557       np->ofile[i] = filedup(curproc->ofile[i]);
558 
559   np->cwd = idup(curproc->cwd);
560   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
561   pid = np->pid;
562   acquire(&ptable.lock);
563   np->state = RUNNABLE;
564   release(&ptable.lock);
565   return pid;
566 }
```

Fig. *thread_create()*

We will be using the ***fork()*** call with some modifications to achieve this. All changes made are as follows:

- Line 530-532: Checks if the allocated stack is not null.
- Line 539: Copy ***pgdir*** pointer of parent process to child thread; this ensures that address space of both child thread and parent process is same otherwise in fork(), it allocates different address space to child process (line 193-198).
- Line 544: Mark this child as a thread process, which will be used later in thread_join().

- Line 548: ***eip*** tells the thread where to start execution from; thus making this to point the function passed as argument.
- Line 549-553: ***esp*** points to top of stack; thus making this to point at top of allocated stack passed as argument. After this, we add ***arg*** and fake PC to the stack which allows thread to exit in case some error occurs during execution.

2. ***thread_join()***

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

Function definition:

int thread_join(void)

```
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
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

Fig. *wait()*

```
568 int
569 thread_join(void)
570 {
571   struct proc *p;
572   int havekids, pid;
573   struct proc *curproc = myproc();
574
575   acquire(&ptable.lock);
576   for (;;) {
577     // Scan through table looking for exited children.
578     havekids = 0;
579     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
580       if (p->is_thread == 0 || p->parent != curproc)
581         continue;
582       havekids = 1;
583       if (p->state == ZOMBIE) {
584         // Found one.
585         pid = p->pid;
586         //kfree(p->kstack);
587         p->kstack = 0;
588         //freevm(p->pgdir);
589         p->pid = 0;
590         p->parent = 0;
591         p->name[0] = 0;
592         p->killed = 0;
593         p->state = UNUSED;
594         release(&ptable.lock);
595         return pid;
596       }
597     }
598     // No point waiting if we don't have any children.
599     if (!havekids || curproc->killed) {
600       release(&ptable.lock);
601       return -1;
602     }
603
604     // Wait for children to exit.  (See wakeup1 call in pr
605     sleep(curproc, &ptable.lock);  //DOC: wait-sleep
606   }
607 }
```

Fig. *thread_join()*

We will be using the ***wait()*** call with some modifications to achieve this. All changes made are as follows:

- Line 580: We need to check if current process is a thread before continuing.
- Line 586: We do not want to free the stack of the thread since parent process may still use that memory.
- Line 588: We do not want to free the complete page allocated to thread since it was same as that of parent process and parent may still use it. In case of wait(), we allocated new space for the child, hence we need to free it.

3. ***thread_exit()***

This call allows a thread to terminate.

Function definition:

int thread_exit(void)

```c
void
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");
}
```

Fig. *exit()*

```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");
}
```

Fig. *thread_exit()*

We will be using ***exit()*** call to achieve this. No changes required here because of a thread is exactly similar to that of a child process. This just passes the control back to parent process.

The above three functions are declared in file **proc.c**. Now we have to create system calls for each of these functions.

First, we make corresponding functions in **sysproc.c** file.

```c
// Create a new kernel thread
int
sys_thread_create(void)
{
  // These functions do not take arguments directly
  // Hence using these lines, we read arguments.
  void (*fcn)(void*), *arg, *stack;
  argptr(0, (void*) &fcn, sizeof(void (*)(void *)));
  argptr(1, (void*) &arg, sizeof(void *));
  argptr(2, (void*) &stack, sizeof(void *));
  return thread_create(fcn, arg, stack);
}
```

```c
// Wait for child thread to complete
int
sys_thread_join(void)
{
  return thread_join();
}
// Terminate a kernel thread
int
sys_thread_exit(void)
{
  return thread_exit();
}
```

Now, we declare them in various files (similar to all other system calls):

1. syscall.c
2. syscall.h
3. user.h
4. usys.S
5. defs.h

Now to test our implementation, we will create a file **thread.c** which will make use of our newly created system calls to perform a task. Next, we have to update **Makefile** and include **thread** in UPROGS list to add this as a command line instruction in XV6 kernel which will execute **thread.c**.

```
$ thread
SSttaratrintg idon_gw odrok_:w so:rbk2
: s:b1
Done s:2F9C
Done s:2F78
Threads finished: (13):14, (14):13, shared balance:3207
$ thread
SSttaartrtiinngg  ddoo__wwoorrkk::  ss::bb2
1
Done s:2F9C
Done s:2F78
Threads finished: (16):17, (17):16, shared balance:3204
$ thread
SSttaarrtitinngg  ddoo_w_owrokr: k:s: s:b2b1

Done s:2F9C
Done s:2F78
Threads finished: (19):20, (20):19, shared balance:3203
```

Upon execution of **thread**, we got this output. We can clearly see that **value of total_balance is not equal to expected 6000**, i.e., sum of all account's balance. This happens because both threads may read an old value of the total_balance at the same time due to context switching and then update it at almost the same time. As a result, the increment of the balance from one of the threads is lost.
Also, on running it multiple times, we can see that value of total_balance keeps changing.

Part 2: Synchronization

To fix the issue of synchronization faced in last case, we have to use **spinlocks** that will allow update to happen atomically. To do so, we have to create a lock data structure called ***thread_spinlock*** and declare three functions:

1. ***void thread_spin_init(struct thread_spinlock *lk)*** to initialize the lock.
2. ***void thread_spin_lock(struct thread_spinlock *lk)*** to acquire the lock.
3. ***void thread_spin_unlock(struct thread_spinlock *lk)*** to release the lock.

```c
struct thread_spinlock{
        volatile uint lock;
};

void thread_spin_init(struct thread_spinlock *lk) {
    lk->lock = 0;
};

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

};

void thread_spin_unlock(struct thread_spinlock *lk) {
    __sync_synchronize();
    asm volatile("movl $0, %0" : "+m" (lk->lock) : );
};
```

- The ***thread_spin_init()*** function just initializes the value of variable ***lock*** to ***0***.

To acquire the lock, we used ***xchg*** function present in ***x86.h*** file which is basically a **compare and swap** assembly level instruction. It returns the current value of lock and if it is 0, it updates it to 1 but returns old value only.

The **Volatile** keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

```c
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

__sync_synchronise() function is used to synchronize data in all threads, i.e., the value of lock is updated in all threads simultaneously.

- The ***thread_spin_lock()*** function will check if lock is acquired or not. If not, it will acquire the lock and synchronize it across all threads.
  If lock == 1, it means it is acquired by some thread, else if lock == 0, it is ready to be acquired by some thread.
- The ***thread_spin_unlock()*** function releases the lock by making its value equal to 0 via assembly level code.

After this, we copy ***thread.c*** to use spinlock around the critical section of code to new file called ***thread_spinlock.c***. And similarly to last part, update ***Makefile*** to add this in command line instruction of Xv6 kernel.

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

First, acquire the lock, then read the old value, increment it and write back the update value, thereafter releasing the lock.

```
$ thread_spinlock
SStatarrttiinngg  do_dwoork_: wso:rkb:2 s
:b1
Done s:2F8C
Done s:2F68
Threads finished: (4):5, (5):4, shared balance:6000
$ thread_spinlock
SSttaratirngt dino_g wdoor_kw:o sr:k:b 1s
:b2
Done s:2F8C
Done s:2F68
Threads finished: (7):8, (8):7, shared balance:6000
$ thread_spinlock
SStatarrttiinngg d o_dwoor_wko:rk s:: bs2
:b1
Done s:2F8C
Done s:2F68
Threads finished: (10):11, (11):10, shared balance:6000
```

Upon execution of thread_spinlock in Xv6, we can see that no matter how many times we run it, the **value of total_balance is 6000** that is equal to expected one.

This happens because spinlock does not allow another thread to enter critical section and read/update the value of total_balance unless the lock is released. Only after releasing the lock, the second thread will be able to read/update this.

Hence, increment from both threads are reflected in final value.

Spinlocks that we have implemented might be inefficient in some cases. When all threads of the process run in parallel on different CPUs, spinlocks are perfect. However, if system has single physical CPU or it is under high load and a context switch occurs in a critical section, then all threads of the process start to spin endlessly, waiting for the lock-holding thread to be release lock.
One possible approach is to implement a different synchronization primitive, a **mutex**.
Mutex is similar to spinlock except that **spinlock causes a thread trying to acquire it to simply wait in the loop and repeatedly check for its availability** while **mutex is a program object that is created so that multiple processes can take turns sharing the same resource**, i.e., the CPU time is not wasted checking for the condition of while loop as process goes to sleep.

To do so, we have to create a lock data structure called ***thread_mutex*** and declare three functions:

1. ***void thread_mutex_init(struct thread_mutex *m)*** to initialize the lock.
2. ***void thread_mutex_lock(struct thread_mutex *m)*** to acquire the lock.
3. ***void thread_mutex_unlock(struct thread_mutex *m)*** to release the lock.

```
23  struct thread_mutex{
24          volatile uint lock;
25  };
26
27  void thread_mutex_init(struct thread_mutex *m) {
28      m->lock = 0;
29  };
30
31  void thread_mutex_lock(struct thread_mutex *m) {
32      while (xchg(&m->lock, 1) != 0)
33          sleep(1);
34      __sync_synchronize();
35  };
36
37  void thread_mutex_unlock(struct thread_mutex *m) {
38      __sync_synchronize();
39      asm volatile("movl $0, %0" : "+m" (m->lock) : );
40  };
```

The implementation of this is exactly similar to that of spinlock except for **thread_mutex_lock** function in which we add ***sleep(1)*** in the while loop (line 33).

This allows the process to go in sleep if lock is not available so that the CPU can meanwhile finish other tasks instead of continuously checking for while loop condition.

```
thread_mutex_lock(&ml);
old = total_balance;
delay(100000);
total_balance = old + 1;
thread_mutex_unlock(&ml);
```

After this, we created a new file **thread_mutex.c** that uses mutex instead of spinlock in the test code and updated **Makefile** to add this as another command line instruction.

```
$ thread_mutex
SSttaarrtitingn gd od_o_wworork:k:  ss::bb21

Done s:2F8C
Done s:2F68
Threads finished: (5):6, (6):5, shared balance:6000
$ thread_mutex
SSttaratritnign g ddoo__wwoorrkk:: s s::bb21

Done s:2F68
Done s:2F8C
Threads finished: (8):8, (9):9, shared balance:6000
```

As expected, this time also, we got expected results, i.e., the final value of **total_balance** is 6000.

Key Differences between Spinlock and Mutex:

Spinlock	Mutex
It is a type of lock that causes a thread attempting to obtain it to check for its availability while waiting in a loop continuously.	It is a program object designed to allow different processes to take turns sharing the same resource.
The process may not sleep while waiting for the lock.	The process may sleep while waiting for the lock.
Spinlock makes no use of context switching.	It involves context switching.
It temporarily prevents a thread from moving.	It may block a thread for an extended amount of time.
It is useful for limited critical sections; else, it wastes the CPU cycles.	It is useful for crucial extended areas where frequent context switching would add overhead.
It does not support preemption.	It supports preemption.

Submission:

We created patch file using the command: ***diff -ruN xv6-public xv6-public-assign1 > patchfile.patch***

The submitted zip folder ***C21.zip*** contains:

1. patchfile.patch
2. Report.pdf
3. xv6-public-assign1: The xv6 directory containing the complete code for the assignment in-case the patch file does not work.