OS-344 Assignment-1

Instructions

- **This assignment is prerequisite for all the other assignments that follow.**
- **Assignment has to be done individually and a report, with relevant screenshots/ necessary observations has to be submitted.**
- **We expect a sincere and fair effort from your side. All submissions will be checked for plagiarism through a software and plagiarized submissions will be penalized heavily, irrespective of the source and copy.**
- **There will be a viva associated with assignment. Attendance of all group members is mandatory.**
- **Assignment code, report and viva all will be considered for grading.**
- **Early start is recommended, any extension to complete the assignment will not be given.**

Kernel Threads and Synchronization

This homework asks you to first implement kernel threads and then build ***spinlocks*** and ***mutexes*** to synchronize access among them. To make it more real and fun, we will pretty much implement interface of the POSIX threads that are de facto standard on most UNIX systems.

You will program the xv6 operating system, so you should use the same setup as you had used previously.

Specifically, you'll 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()***. Then, you'll implement POSIX-like synchronization primitives: ***spinlocks*** and ***mutexes***. To test your implementation you will use a simple program that we provide.

Part 1: Kernel threads

Before starting to work on your thread implementation, you should understand what threads are. Here is a good link that introduces POSIX threads that you have to develop (you don't have to read all of it, just get the basic idea of what threads are and how they work): ***pthread_tutorial***. If you feel like it you can also read a couple of chapters from the OSTEP book: ***Concurrency: An Introduction*** and ***Interlude: Thread API***

The take-away idea for threads: threads are very much like processes (they can run in parallel on different physical CPUs), but 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.

While threads share the same address space they each need their own stack as they might execute entirely different code in the program (call different functions with different arguments --- all this information has to be preserved for each thread individually, hence they need different stacks. The parent process allocates the stacks with ***malloc()*** or ***sbrk()*** for each thread before starting it (obviously, this can be hidden inside the ***thread_create()*** function). While these new stacks will not have a guard page in front of it, but otherwise should work just fine (the heap is mapped with the same attributes as stack (writable)).

Now, lets get back to work. Your new ***thread_create()*** system call should look like this:

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

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.

The other new system call is ***int thread_join(void)***. 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.

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

You also need to think about the semantics of a couple of existing system calls. For example, ***int wait()*** should wait for a child process that does not share the address space with this process. It should also free the address space if this is last reference to it. Finally, ***exit()*** should work as before but for both processes and threads; little change is required here.

You can follow the example template for thread.c to test your thread implementation:

```
#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();
}
```

Here the process creates two threads that execute the same ***do_work()*** function concurrently. The ***do_work()*** function in both threads updates the shared variable ***total_balance***.

Hints:

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.

Part 2: Synchronization

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 increament of the balance) from one of the threads is lost.

Spinlocks

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 un-comment 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).

Mutexes

While spinlocks that you'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 you are running on a system with a single physical CPU (you can change the number of CPUs in the xv6 Makefile and set it to 1), or the system is under high load and a context switch occurs in a critical section (you can imagine that it can happen in a slightly longer critical section) then all threads of the process start to spin endlessly, waiting for the interrupted (lock-holding) thread to be rescheduled and run again the spinlocks become inefficient.

One possible approach is to implement a different synchronization primitive, a mutex, and instead of spinning on a thread release the CPU to another thread, like:

```
void thread_mutex_lock(struct thread_mutex *m)
{
  while(locked(m))
    yield();
}

void thread_mutex_unlock(struct thread_mutex *m)
{
  unlock(m);
}
```

Based on the high-level description of the mutex above, implement a mutex that will allow you to execute the update atomically similar to spinlock, but instead of spinning will release the CPU to another thread. Test your implementation by replacing spinlocks in your example above with mutexes.

Specifically you should define a simple mutex data structure and implement three functions that: **1)** initialize the mutex to the correct initial state ***(void thread_mutex_init(struct thread_mutex *m))***, ***2)*** a function to acquire a mutex ***(void thread_mutex_lock(struct thread_mutex *m))***, and ***3)*** a function to release it ***void thread_mutex_unlock(struct thread_mutex *m)***.

Mutexes can be implemented very similarly to spinlocks (the implementation you already have). Since xv6 doesn't have an explicit ***yield(0)*** system call, you can use ***sleep(1)*** instead.

Submission instructions

- Place the code files and report file into a zip folder and name it with your Group-No, say [**C1.zip**]
- Also create a patch file for the all the edited code, which we can directly apply on the xv6 directory and it will be able to update the code files with new ones.
- Report.pdf should contain a detailed description of all of your understanding about the given set of questions including screenshots while performing different given tasks.
- Further, you must describe your test cases in some detail, and the observations you made from them.
- Only one member of the group should submit the assignment

--End of Assignment--