Home

HW6: POSIX Threads

This homework asks you to extend xv6 with support for POSIX threads. To make it real and fun, we pretty much implement the 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 for the previous homework.

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()`.

As a challenge exercise you can implement POSIX-like synchronization primitives: spinlocks and mutexes. To test your implementation you will use a simple program we provide.

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): https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html. 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 the all 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 following 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:%s\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.

Extra credit: 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 increment of the balance) from one of the threads is lost.

Extra credit (5%): 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 uncomment 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).

Extra credit (10%): 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 Makefule 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 scheduled 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 funtion 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.

Extra credit (15%): Conditional variables

While spinlock and mutex synchronization work well, sometimes we need a synchronization pattern similar to the producer-consumer queue we've discussed in class, i.e., instead of spinning on a spinlock or yielding the CPU in a mutex, we would like the thread to sleep until certain condition is met.

POSIX provides support for such scheme with conditional variables. A condition variable is used to allow a thread to sleep until a condition is true. Note that conditional variables are always used along with the mutex.

You have to implement conditional variables similar to the once provided by POSIX. The function primarily used for this is pthread_cond_wait(). It takes two arguments; the first is a pointer to a condition variable, and the second is a locked mutex. When invoked, pthread_cond_wait() unlocks the mutex and then pauses execution of its thread. It will now remain paused until such time as some other thread wakes it up. These operations are "atomic;" they always happen together, without any other threads executing in between them.

```
struct q {
   struct thread_cond cv;
   struct thread_mutex m;

   void *ptr;
};

// Initialize
```

```
thread_cond_init(&q->cv);
thread_mutex_init(&q->m);

// Thread 1 (sender)
void*
send(struct q *q, void *p)
{
   thread_mutex_lock(&q->m);
   while(q->ptr != 0)
      ;
   q->ptr = p;
   thread_cond_signal(&q->cv);
   thread_mutex_unlock(&q->m);
}

// Thread 2 (receiver)

void*
recv(struct q *q)
{
  void *p;

  thread_mutex_lock(&q->m);

  while((p = q->ptr) == 0)
    pthread_cond_wait(&q->cv, &q->m);
  q->ptr = 0;

  thread_mutex_unlock(&q->m);
  return p;
}
```

To test your solution, develop a small queue example like above and submit it along with your extra credit solution.

Extra credit (5%): Semaphores

Conditional variables can be used to implement semaphores (if you are still confused about semaphores read about them here: Semaphore (programming)). The implementation of the semaphore is trivial and you can read up on it here and implement it in a similar manner: Condition Variables.

Implement semaphores and the producer consumer queue of N elements in which access to the queue is controlled with semaphores like described here: Semaphore (programming).

Submit

Please submit code to Gradescope Gradescope. You have to submit a compressed (zipped) archive of your xv6-public folder. All

the functions should be available for user programs and can be used by including "user.h" file. You can resubmit as many times as you wish. The structure of the zip file should be the following:

```
./xv6-public
   | - all xv6 files
```

Updated: April, 2023