January 2023 CSE 314

Offline Assignment 5: xv6 - Threading & Synchronization

Deadline: Monday , August 21, 2023, 11:45 PM

It is your duty to be up to date with the spec.

Change logs

Change	Time of modification

Introduction

In our previous offline, we used **POSIX** threads to solve *synchronization problems*. As smart CSE students, we won't be limited to a user only, but also design thread and synchronization primitives of our own. In this offline, we will add support for threads in **xv6**. We will implement a user-level thread library consisting of some system calls related to threads that are very familiar to us. Then, we will implement **POSIX-like** synchronization primitives in **xv6**. Finally, we will use these primitives to solve the **producer-consumer** problem.

Background

- Revisit the difference between *process* and *thread*. The main difference is that threads share the same address space, while processes have their own address space.
- Thoroughly understand what threads are and how you used them in your assignment on **IPC**. pthreads
- Key takeaway 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).
- Though the threads share a common address space, each thread requires its own stack. This is because each thread 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)

Task 1: Implementing thread support in xv6

You need to write three system calls * `thread_create()` * `thread_join()` * `thread_exit()`

Our first system call `thread_create()` would have the following signature:

```
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.

The new thread creation will be similar to the new process creation. Actually, we will create a new process with the same address space as the calling process. The only difference is that we will create a new stack for the new 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 **strictly one-page** in size. The new thread starts executing at the address specified by `fcn`.

The other new system call is:

```
int thread_join(int thread_id)
```

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

The last system call is:

```
void thread_exit(void)
```

To test your implementation you may use the following program `threads.c`:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/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( "Starting do_work: s:%s\n", b->name);
```

```
    for (i = 0; i < b->amount; i++) {
        // lock and mlock will be implemented by you.
         // thread_spin_lock(&lock);
         // thread_mutex_lock(&mlock);
         old = total_balance;
         delay(100000);
         total_balance = old + 1;
     // if(old + 1 != total_balance)  printf("we missed an update. old: %d
total_balance: %d\n", old, total_balance);
         //thread_spin_unlock(&lock);
         // thread_mutex_lock(&mlock);

    }

    printf( "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 thread1, thread2, r1, r2;

  s1 = malloc(4096); // 4096 is the PGSIZE defined in kernel/riscv.h
  s2 = malloc(4096);

  thread1 = thread_create(do_work, (void*)&b1, s1);
  thread2 = thread_create(do_work, (void*)&b2, s2);

  r1 = thread_join(thread1);
  r2 = thread_join(thread2);

  printf("Threads finished: (%d):%d, (%d):%d, shared balance:%d\n",
      thread1, r1, thread2, r2, total_balance);

  exit();
}
```

Make necessary changes if required. Here we create two threads that execute the same `do_work()` function concurrently. The do_work() function in both threads deposits(updates) the shared variable total_balance.

Structure proc may need some more updates,

```
struct proc {
    // previous ones
```

```
    struct spinlock memlock;  // find places to set and release the locks
    int is_thread;              // if it is thread
    int mem_id;                  // All threads will have the same physical
  pages with the mothrer, hence the same memory ID
  };
```

Hints

The `thread_create()` call should behave very much like a fork, except that instead of copying the address space to a new page table, it initialises the new process so that the new process and cloned process use the same page table. Thus, memory will be shared, and the two "processes" are really threads. You have to think about returning to function `fcn` after thread creation. Please study `p->trapframe->epc` for this task. You also have to replace the user stack with the supplied stack. Look more closely at `p->trapframe->sp`. Find your own way out to copy the `arg` to stack to make it available to function `fcn()` . `kernel\exec.c` can be a good example to follow.

previous memory layout	current memory layout

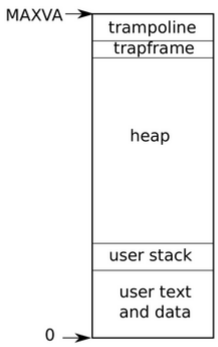
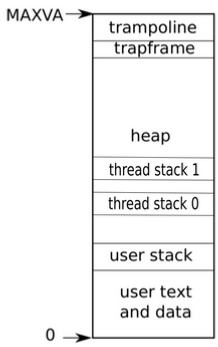


Figure 2.3: Layout of a process's virtual address space



Layout of a process's virtual address space with 2 threads.

Please understand the trapframe page in Figure 2.3. As the thread uses the same page table, how do we map its trapframe page? Do we need to consider any more pages? And the very first thing we did, making the page tables exactly equal, was it a wise decision? Now, it may be a good time to visit Chapter 2 from the xv6 book.

Understand what `uvmcopy()` is doing in `fork()`. Plan for a similar but slightly different `uvmmirror()` approach that doesn't use `kalloc()` i.e. doesn't allocate new physical pages. `mappages()` should be a good friend of you. One last thing that you have to brainstorm, is how to keep the page tables synchronized when a new page is allocated or deallocated by any of the threads.

The `int thread_join(int thread_id)` system call is very similar to the already existing `int wait(uint64 addr)` 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. Please understand how `exit()` works and the necessity of reparenting.

Task 2: Implementing synchronization primitives in xv6

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 threas. 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. Try uncommenting the **printf** part.

Spinlock

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(`struct thread_spinlock`) and implement three functions:

1. initialize the lock to the correct initial state (`void thread_spin_init(struct thread_spinlock *lk)`)
2. a function 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)`

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

Suppose, you are running on a system with a **single** physical CPU, or the system 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 interrupted (lock-holding) thread to be scheduled and run again the spinlocks become inefficient. If you look closely, the main culprit is that the threads are spinning in a loop, wasting CPU cycles. **Mutexes to our rescue!**

A higher-level pseudo-code for a mutex is as follows:

```
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 (`struct thread_mutex` and implement three functions:

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)`)
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 from the `thread_mutex_lock` function or may design a new system call which will call yield.

Conditional Variables

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

POSIX provides support for such a 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 ones 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 the 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.

Semaphores

Conditional variables can be used to implement semaphores. Follow this to implement semaphores using conditional variables.

A sample program `producer_consumer.c` to test your semaphore implementation is given below:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

struct queue{
    int arr[16];
    int front;
    int rear;
    int size;
    queue()
    {
```

```
        front = 0;
        rear = 0;
        size = 0;
    }
    void push(int x)
    {
        arr[rear] = x;
        rear = (rear+1)%16;
        size++;
    }
    int front()
    {
        if(size==0)
            return -1;
        return arr[front];
    }
    void pop()
    {
        front = (front+1)%16;
        size--;
    }
};
struct queue q;
// a mutex object lock
// a semaphore object empty
// a semaphore object full

void init_semaphore()
{
    // initialize mutex lock
    // initialize semaphore empty with 5
    // initialize semaphore full with 0

}

void * ProducerFunc(void * arg)
{
    printf("%s\n",(char*)arg);
    int i;
    for(i=1;i<=10;i++)
    {
        // wait for semphore empty

        // wait for mutex lock

        sleep(1);
        q.push(i);
        printf("producer produced item %d\n",i);

        // unlock mutex lock
        // post semaphore full
    }
}
```

```
void * ConsumerFunc(void * arg)
{
    printf("%s\n",(char*)arg);
    int i;
    for(i=1;i<=10;i++)
    {
        // wait for semphore full
        // wait for mutex lock


        sleep(1);
        int item = q.front();
        q.pop();
        printf("consumer consumed item %d\n",item);


        // unlock mutex lock
        // post semaphore empty
    }
}

int main(void)
{

    init_semaphore();

    char * message1 = "i am producer";
    char * message2 = "i am consumer";


    void *s1, *s2;
    int thread1, thread2, r1, r2;

    s1 = malloc(4096);
    s2 = malloc(4096);

    thread1 = thread_create(ProducerFunc, (void*)message1, s1);
    thread2 = thread_create(ConsumerFunc, (void*)message2, s2);

    r1 = thread_join();
    r2 = thread_join();

    exit();
}
```

Please make the necessary changes to make it run.

Mark Distribution

Task	Mark

	Task	Mark
1	Proper implementation of Task 1	45
2	Spinlock implementation	10
3	Mutex implementation	15
4	Conditional Variable implementation	20
5	Semaphore implementation	10

Please make sure your `threads.c` and `producer_consumer.c` is running. This will make the evaluation process faster.

Submission guideline

Start with a fresh copy of xv6-riscv from the original repository. Make necessary changes for this offline. In this offline, you will submit just the changes done (i.e.: a patch file), not the entire repository. Don't commit. Modify and create files that you need to. Then create a patch using the following command:

```
git add --all
git diff HEAD > {studentID}.patch
```

Where studentID = your own seven-digit student ID (e.g., 1905XXX). Just submit the patch file, do not zip it. In the lab, during evaluation, we will start with a fresh copy of xv6 and apply your patch using the command: git apply {studentID}.patch Make sure to test your patch file after submission in the same way we will run it during the evaluation.

Special Instructions

1. This offline is a very complex one. So please start early.
2. **Discussion with peers** is encouraged when you are stuck (Specially with the **panics** you will face).** A direct copy will be strictly punished.**
3. You are **encouraged** to explore codes from github. But whatever you do, please make sure to **understand it fully**.
4. You must do the offline on top of `xv6-riscv`.
5. Coding for more than 1:30 hours daily (for this offline) is not recommended.

Some resources which might be helpful:

1. https://pages.cs.wisc.edu/~gerald/cs537/Summer17/projects/p4b.html
2. https://courses.cs.duke.edu/fall22/compsci310/thread.html
3. https://moss.cs.iit.edu/cs450/mp4-xv6.html
4. https://www.youtube.com/watch?v=1c3Bd8NlkIQ&ab_channel=TheSparkle
5. https://www.youtube.com/watch?v=0zMYKwo482c&t=2464s&ab_channel=TheSparkle
6. https://users.cs.utah.edu/~aburtsev/238P/2018winter/hw/hw4-threads.html
7. https://pdos.csail.mit.edu/6.S081/2020/labs/thread.html