Lab 3 Report

Christina Pavlopoulou
cpavl001@ucr.edu

Niloufar Hosseini Pour
nhoss003@ucr.edu

Andres Calderon
acald013@ucr.edu

March 16, 2016

The goal of this lab was to create kernel threads in `xv6`.

1 Clone

In this section, we describe our implementation of the system call `clone` that we created. Firstly, we added this function in the `proc.c` file of `xv6` (see listing 1). We based on the `fork()` implementation but we changed the following things:

1. Instead of different address spaces between the parent and the child, in `clone` the child process has the same address space with the parent.
2. In `clone`, the user stack is different for the child process than the one that the parent uses.
3. We, also, need to change the trap frame because we have to point our new registers to the new user stack. So, when we go back to user space we need the registers to be restored. As a result, we need to change the base pointer and the stack pointer.
4. Finally, we change the `wait()` function in `proc.c` (listing 2). The parent process releases the resources when the child process is finished. But now more threads share the same resources. So, we added a parameter that keeps track of the thread that the system currently releases.

2 Thread Library

In this section, we created a file called `thread.h` (see listing 3). In this file, we added `init_lock()`, `acquire_lock()` and `release_lock()` function in the same way as in the `spinlock.c` file. The only difference is that we did not use the CPUs and the name variables because we did not need them for our implementation.

In the `init_lock` we just initialize the locked variable. In the `acquire_lock`, we spin the threads that do not have the lock until the thread with the lock calls the `release_lock` and release it. Finally, in the same file, we implement the `create_thread` function in which we define the size of our stack and we allocate space for it. Then, we call the `clone` system call. As final step, we free the stack and we exit.

```c
int clone(void){
  char *ustack;
  int i, pid;
  int size;
  struct proc *np;

  if((np = allocproc()) == 0)
    return -1;

  //We take the arguments for size and user stack
  argint(1,&size);
  argptr(0,&ustack, size);

  //We share the same address space
  np->pgdir = proc->pgdir;
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;

  // We align the address stack
  ustack = (char*)PGROUNDUP((uint)ustack);

  // We point the esp and ebp pointers to the new user stack
  np->tf->esp = (uint)ustack + (proc->tf->esp - PGROUNDDOWN(proc->tf->esp));
  np->tf->ebp = (uint)ustack + (proc->tf->ebp - PGROUNDDOWN(proc->tf->ebp));

  // We calculate the number of addresses we have to copy to new user stack
  uint usize = size - (proc->tf->esp - PGROUNDDOWN(proc->tf->esp));
  // We copy them
  memmove((void *)(np->tf->esp), (const void *)(proc->tf->esp), usize);
  // We return -1 in the clone processes
  np->tf->eax = -1;

  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);
  np->cwd = idup(proc->cwd);

  safestrcpy(np->name, proc->name, sizeof(proc->name));

  pid = np->pid;
  // We mark the PID of the clone process
  np->pid = 0;

  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);

  return pid;
}
```

Listing 1: Clone function for thread creation.

```c
    // We free the address space just if it is not a clone process
    if(pid > 0){
      freevm(p->pgdir);
    }
```

Listing 2: Changes in the `wait` function.

```c
#include "types.h"
#include "user.h"
#include "x86.h"

int n;

struct lock {
  uint locked;
};

int thread_create2(void *(*start_routine)(void*), void *arg){
  int size = 4096;
  char *stack = malloc(2 * size);

  int tid = clone(stack, size);
  if(tid == -1){
    (*start_routine)(arg);
    free(stack);
    exit();
  }
  return tid;
}

void init_lock(struct lock *lk) {
  lk->locked = 0;
}

void acquire_lock(struct lock *lk){
  while(xchg(&lk->locked, 1) != 0);
}

void release_lock(struct lock *lk){
  xchg(&lk->locked, 0);
}
```

Listing 3: Library for thread creation and locking.

3 Anderson's array-based queue lock

3.1 Initialization

First we used a new struct and dynamically create an array and make all the cells equal to zero except the first cell.

```
struct anderson_lock {
  uint *slots;
  uint next_slot;
};

void init_anderson_lock(struct anderson_lock *lk, uint nthreads){
  lk->slots = malloc(nthreads*sizeof(uint));
  int i;
  for(i = 0; i < nthreads; i++) lk->slots[i] = 0;
  lk->slots[0] = 1;
  lk->next_slot = 0;
}
```

Listing 4: Initializing an Anderson lock.

3.2 Acquiring the Lock

Threads share an atomic tail field, to acquire the lock, each thread atomically increments the tail field. If the flag is true, the lock is acquired , otherwise, spin until the flag is true. If another thread wants to acquire the lock, it applies get and increment,the thread spins because the flag is false.

```
uint acquire_anderson_lock(struct anderson_lock *lk){
  uint myplace;
  xchg(&myplace, lk->next_slot);
  xchg(&lk->next_slot, lk->next_slot + 1);
  myplace = myplace % n;
  while(lk->slots[myplace] == 0);
  return myplace;
}
```

Listing 5: Acquiring an Anderson lock.

3.3 Releasing the Lock

The first thread releases the lock by setting, the next slot to true, the second thread notices the change and gets the lock.

```
void release_anderson_lock(struct anderson_lock *lk, uint myplace){
  xchg(&lk->slots[myplace % n], 0);
  xchg(&lk->slots[(myplace + 1) % n], 1);
}
```

Listing 6: Releasing an Anderson lock.

4 Tests

4.1 Spin lock test

The first of our tests implements the simple spin lock explained in section 2. We used a fix number of threads (4 in this case) and a counter that we put between the acquire and release functions to test that the threads are interleaving and increment the counter appropriately. For each thread we run n iterations, where n is the number of threads. The code can be seen in listing 7. Figure 1 shows the output of this test.

```
#include "thread.h"

int x = 1;
int nthreads = 4;
struct lock our_lock;

void *my_function(void *arg){
  int i;
  int j = (int)arg;
  for(i = 0; i < nthreads; i++){
    acquire_lock(&our_lock);
      printf(1,"I am thread %d, Let's see x = %d.\n", j, x);
      x = x + 1;
    release_lock(&our_lock);
    sleep(10);
  }
  return 0;
}

int main(int argc, char *argv[]){
  init_lock(&our_lock);

  int i;
  for(i = 0; i < nthreads; i++){
    thread_create2(*my_function, (int*)i);
  }
  for(i = 0; i < nthreads; i++){
    wait();
  }

  exit();
}
```

Listing 7: Spin lock test.

4.2 Anderson lock test

Here we implement the frisbee game. We ask the number of threads and number of passes as parameters. We allocate an Anderson lock and create the requested number of threads. Then, we run a function with an infinite loop which stops when the number of passes is reached. Listing 8 shows our implementation. The output of the test can be seen in figure 2

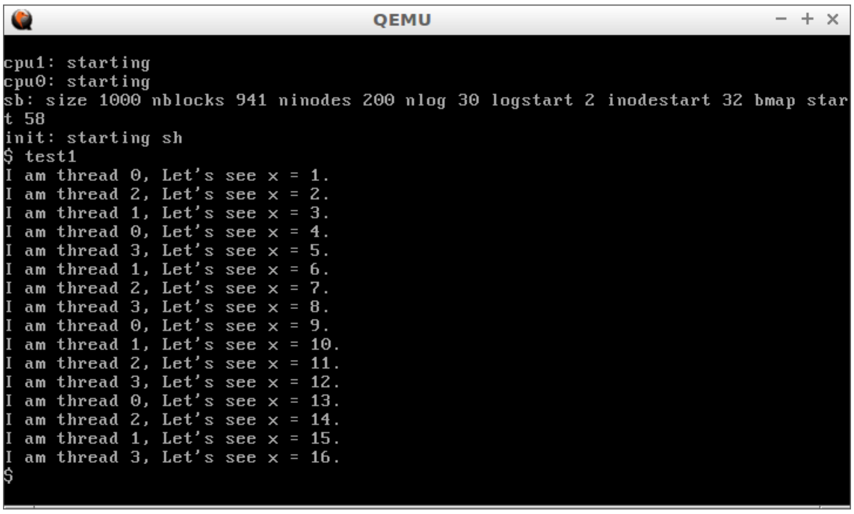

Figure 1: Output of test 1. Number of threads was set to 4.

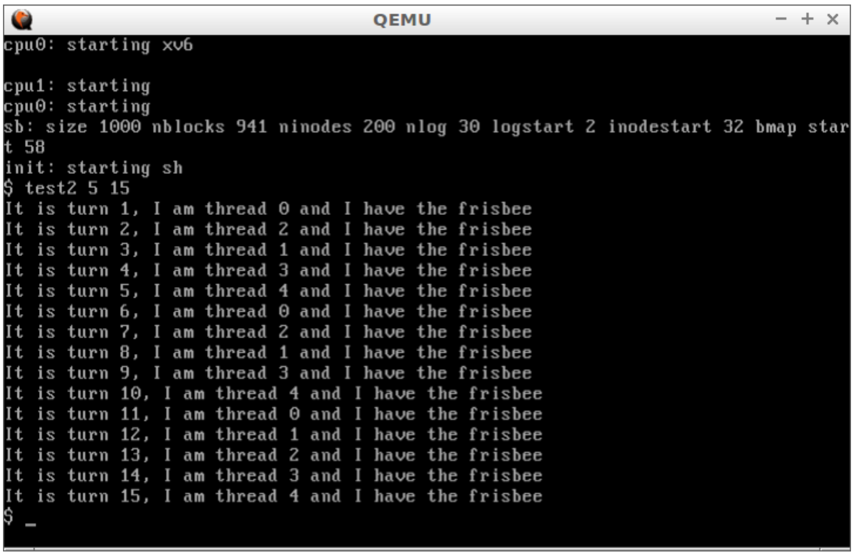

Figure 2: Output of test 2. Frisbee game runs for 5 threads during 15 passes.

```c
#include "thread.h"

int x = 1;
int stop;
struct anderson_lock our_lock;

void *my_function(void *arg){
  int j = (int)arg;
  int k;
  for(;;){
    k = acquire_anderson_lock(&our_lock);
      if(x > stop){
        release_anderson_lock(&our_lock, k);
        break;
      }
      printf(1,"It is turn %d, I am thread %d and I have the frisbee...\n", x, j);
      x = x + 1;
    release_anderson_lock(&our_lock, k);
  }
  return 0;
}

int main(int argc, char *argv[]){
  n = atoi(argv[1]);
  stop = atoi(argv[2]);
  init_anderson_lock(&our_lock, n);

  int i;
  for(i = 0; i < n; i++){
    thread_create2(*my_function, (int*)i);
  }
  for(i = 0; i < n; i++){
    wait();
  }

  exit();
}
```

Listing 8: Anderson lock test.