

# 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.

```

169 int clone(void){
170     char *ustack;
171     int i, pid;
172     int size;
173     struct proc *np;
174
175     if((np = allocproc()) == 0)
176         return -1;
177
178     //We take the arguments for size and user stack
179     argint(1,&size);
180     argptr(0,&ustack, size);
181
182     //We share the same address space
183     np->pgdir = proc->pgdir;
184     np->sz = proc->sz;
185     np->parent = proc;
186     *np->tf = *proc->tf;
187
188     // We align the address stack
189     ustack = (char*)PGROUNDDUP((uint)ustack);
190
191     // We point the esp and ebp pointers to the new user stack
192     np->tf->esp = (uint)ustack + (proc->tf->esp - PGROUNDDOWN(proc->tf->esp));
193     np->tf->ebp = (uint)ustack + (proc->tf->ebp - PGROUNDDOWN(proc->tf->ebp));
194
195     // We calculate the number of addresses we have to copy to new user stack
196     uint usize = size - (proc->tf->esp - PGROUNDDOWN(proc->tf->esp));
197     // We copy them
198     memmove((void *) (np->tf->esp), (const void *) (proc->tf->esp), usize);
199     // We mark the clone processes with a -1
200     np->tf->eax = -1;
201
202     for(i = 0; i < NOFILE; i++)
203         if(proc->ofile[i])
204             np->ofile[i] = filedup(proc->ofile[i]);
205     np->cwd = idup(proc->cwd);
206
207     safestrcpy(np->name, proc->name, sizeof(proc->name));
208
209     pid = np->pid;
210     np->pid = 0;
211
212     acquire(&ptable.lock);
213     np->state = RUNNABLE;
214     release(&ptable.lock);
215
216     return pid;
217 }

```

Listing 1: Clone function for thread creation.

```

285     // We free the address space just if it is not a clone process
286     if(pid > 0){
287         freevm(p->pgdir);
288     }

```

Listing 2: Changes in the wait function.

```

1  #include "types.h"
2  #include "user.h"
3  #include "x86.h"
4
5  int n;
6
7  struct lock {
8      uint locked;
9  };
10
11 int thread_create2(void *(*start_routine)(void*), void *arg){
12     int size = 4096;
13     char *stack = malloc(2 * size);
14
15     int tid = clone(stack, size);
16     if(tid == -1){
17         (*start_routine)(arg);
18         free(stack);
19         exit();
20     }
21     return tid;
22 }
23
24 void init_lock(struct lock *lk) {
25     lk->locked = 0;
26 }
27
28 void acquire_lock(struct lock *lk){
29     while(xchg(&lk->locked, 1) != 0);
30 }
31
32 void release_lock(struct lock *lk){
33     xchg(&lk->locked, 0);
34 }

```

Listing 3: Library for thread creation and locking.

## 3 Anderson's array-based queue lock

### 3.1 Initialization

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

```
41 void init_anderson_lock(struct anderson_lock *lk, uint nthreads){
42     lk->slots = malloc(nthreads*sizeof(uint));
43     int i;
44     for(i = 0; i < nthreads; i++) lk->slots[i] = 0;
45     lk->slots[0] = 1;
46     lk->next_slot = 0;
47 }
```

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.

```
49 uint acquire_anderson_lock(struct anderson_lock *lk){
50     uint myplace;
51     xchg(&myplace, lk->next_slot);
52     xchg(&lk->next_slot, lk->next_slot + 1);
53     myplace = myplace % n;
54     while(lk->slots[myplace] == 0);
55     return myplace;
56 }
```

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.

```
58 void release_anderson_lock(struct anderson_lock *lk, uint myplace){
59     xchg(&lk->slots[myplace % n], 0);
60     xchg(&lk->slots[(myplace + 1) % n], 1);
61 }
```

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.

```
1  #include "types.h"
2  #include "user.h"
3  #include "thread.h"
4
5  int x = 1;
6  int nthreads = 4;
7  struct lock our_lock;
8
9  void *my_function(void *arg){
10     int i;
11     int j = (int)arg;
12     for(i = 0; i < nthreads; i++){
13         acquire_lock(&our_lock);
14         printf(1, "I am thread %d, Let's see x = %d.\n", j, x);
15         x = x + 1;
16         release_lock(&our_lock);
17         sleep(10);
18     }
19     return 0;
20 }
21
22 int main(int argc, char *argv[]){
23     init_lock(&our_lock);
24
25     int i;
26     for(i = 0; i < nthreads; i++){
27         thread_create2(my_function, (int*)i);
28     }
29     for(i = 0; i < nthreads; i++){
30         wait();
31     }
32
33     exit();
34 }
```

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

```
QEMU
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ test1
I am thread 0, Let's see x = 1.
I am thread 2, Let's see x = 2.
I am thread 1, Let's see x = 3.
I am thread 0, Let's see x = 4.
I am thread 3, Let's see x = 5.
I am thread 1, Let's see x = 6.
I am thread 2, Let's see x = 7.
I am thread 3, Let's see x = 8.
I am thread 0, Let's see x = 9.
I am thread 1, Let's see x = 10.
I am thread 2, Let's see x = 11.
I am thread 3, Let's see x = 12.
I am thread 0, Let's see x = 13.
I am thread 2, Let's see x = 14.
I am thread 1, Let's see x = 15.
I am thread 3, Let's see x = 16.
$
```

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

```
QEMU
cpu0: starting xv6
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ test2 5 15
It is turn 1, I am thread 0 and I have the frisbee
It is turn 2, I am thread 2 and I have the frisbee
It is turn 3, I am thread 1 and I have the frisbee
It is turn 4, I am thread 3 and I have the frisbee
It is turn 5, I am thread 4 and I have the frisbee
It is turn 6, I am thread 0 and I have the frisbee
It is turn 7, I am thread 2 and I have the frisbee
It is turn 8, I am thread 1 and I have the frisbee
It is turn 9, I am thread 3 and I have the frisbee
It is turn 10, I am thread 4 and I have the frisbee
It is turn 11, I am thread 0 and I have the frisbee
It is turn 12, I am thread 1 and I have the frisbee
It is turn 13, I am thread 2 and I have the frisbee
It is turn 14, I am thread 3 and I have the frisbee
It is turn 15, I am thread 4 and I have the frisbee
$ -
```

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

```

1  #include "types.h"
2  #include "user.h"
3  #include "thread.h"
4
5  int x = 1;
6  int stop;
7  struct anderson_lock our_lock;
8
9  void *my_function(void *arg){
10     int j = (int)arg;
11     int k;
12     for(;;){
13         k = acquire_anderson_lock(&our_lock);
14         if(x > stop){
15             release_anderson_lock(&our_lock, k);
16             break;
17         }
18         printf(1, "It is turn %d, I am thread %d and I have the frisbee...\n", x, j);
19         x = x + 1;
20         release_anderson_lock(&our_lock, k);
21     }
22     return 0;
23 }
24
25 int main(int argc, char *argv[]){
26     n = atoi(argv[1]);
27     stop = atoi(argv[2]);
28     init_anderson_lock(&our_lock, n);
29
30     int i;
31     for(i = 0; i < n; i++){
32         thread_create2(*my_function, (int*)i);
33     }
34     for(i = 0; i < n; i++){
35         wait();
36     }
37
38     exit();
39 }

```

Listing 8: Anderson lock test.