# CS202 Lab 3 Report

Md Abdullah Al Mamun *mmamu003@ucr.edu*
Yicheng Zhang *yzhan846@ucr.edu*

## 1 THE LIST OF ALL FILES MODIFIED

In this project, the files that we modified are listed below.

- kernel/syscall.h: Define new system call number for **clone()**.
- kernel/syscall.c: Update system call table for **clone()**.
- kernel/sysproc.c: Define system call function for **clone()**.
- kernel/proc.c: Create a new system call **clone()**. Also modifies exsiting **exit()** and **wait()**.
- kernel/defs.h: Create a new system call **clone()**.
- user/usys.pl and user/user.h: Update user-space syscall interface.
- user/frisbee.c: Create **" frisbee.c"** file in the user directory to test functionality.
- Makefile: Append UPROGS for **frisbee** and **thread.o** in Makefile.

## 2 IMPLEMENTATION OF OUR CLONE() SYSTEM CALL

The system call clone() generates a new thread-like process. In that it produces a child process from the current process, it is quite similar to fork(), however there are a few major differences. The address space of the original process is shared by the child process (fork() produces a new copy of the address space, separating the child from the parent). The stack of the child process is in the original process's address space (fork() creates a separate stack for the child process). clone() takes a function pointer and runs it as soon as the child process starts (fork() starts the child at the same point in the code as the parent). void (*play)(void*) is a pointer to the function that will be executed by the newly formed thread once it has started. When the child executes, void *arg is the argument that will be provided to *play. The memory address void *stack in the parent's address space instructs the child where to put its stack. This permits the child to use the parent's stack address space. The stack address provided must be page-aligned and have at least one page of memory in order to "clone" the current process. We must ensure that both of these prerequisites are fulfilled. If they aren't, we'll have to return a negative value. We added if statements to accomplish this. It conducts an alignment check to verify whether the stack is not aligned, ((uint) stack percent PGSIZE)!= 0. We return -1 if this is true. The second if statement, (currentprocess->sz - (uint) stack) PGSIZE, yields -1 if we have less than 1 page of memory. Details are in Fig. 1 and 2.

To hold the clone's state, we need to create a new process structure. Clone() shared the parent's address space, whereas fork() created a full duplicate of the parent's address space. The pagetable entry in the xv6 process structure records the address space. We do it using np->pagetable = p->pagetable in our code; to share the parent's address space with the clone. That is, the current process curproc and the cloned process np have the same pagetable (e.g., address space).

```
int
clone(void *stack, int size)
{
  int i, pid;
  struct proc *np; // child process
  struct proc *p = myproc(); // parent process

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // child shares the same address space with parent
  np->pagetable = p->pagetable;  // same page table
  np->sz = p->sz;  // same size
  *np->trapframe = *p->trapframe;

  np->trapframe->a0 = 0; // return 0 to child process
```

**Figure 1:** Cloning system call implementation part 1

## 3 THREAD-CREATE() FUNCTION IMPLEMENTATION

Our function void *thread_create(void*(start_routine)(void*),void *arg) takes a function and arg (round number for game) as a parameter. Inside the thread Create function, we set the size of the stack for next child thread. We use clone() to create the child inside this function, and then call start_routine() with the argument arg. We prepare the arguments on the user thread stack. Code implementation is shown in Fig. 3.

## 4 LOCK INITIALIZATION FOR USER LEVEL THREAD

In the user level we used our manual lock. Which initiated by the "lock_init" function. We use lock_acquire for acquiring the lock and finally use lock_release function for releasing the lock. The function named "lock_init" in thread.c initially sets lock->held equal to zero that means nobody holds the lock. cpu equals zero means currently no cpu is holding the lock. this helps us later to track which thread holds the lock and we ensure that a thread cannot hold the same lock which is already acquired by it. Code implementation is shown in Fig. 4.

## 5 LOCK ACQUIRE FOR USER LEVEL THREAD

In order to acquire lock, at first we have to disable interrupts and then prevent that a process which already has acquired a lock but again trying to acquire a lock. If this is not the case then it uses RISC-V atomic exchange to build the spin lock on a shared critical region. No other thread can acquire this lock until current thread releases the lock. Code implementation is shown in Fig. 5.

```
  // use the same file descriptor
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

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

  np->context.sp = (uint64)(stack + PGSIZE - 4);
  *((uint64*)(np->context.sp)-4) = 0xFFFFFFFF;
  np->context.sp = (np->context.sp) - 4;

  pid = np->pid;

  release(&np->lock);

  acquire(&wait_lock);
  np->parent = p;
  release(&wait_lock);

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

  return pid;
}
```

**Figure 2:** Cloning system call implementation part 2

```
void *thread_create(void* start_routine(void*), void* arg) {
  void *t_stack = malloc(8192);
  int tid = clone(t_stack, 8192);

  if (tid == 0){
    (start_routine)(arg);
    return 0;
  }
  texit();
  return 0;
}
```

**Figure 3:** Thread_create function

```
void lock_init(lock_t *l)
{
  l->held = 0;
}
```

**Figure 4:** Lock initialization for user level thread

## 6 LOCK RELEASE FOR USER LEVEL THREAD

while releasing the lock, we first check whether the lock is acquired by the current lock or not. If not then it goes to the panic (terminate XV6 [1] ) mode. If the sanity check is okay, then the CPU releases the lock so that other threads that are waiting for the lock can acquire

```
void lock_acquire(lock_t *l)
{
  while(__sync_lock_test_and_set(&l->held, 1) != 0);
}
```

**Figure 5:** Lock acquire for user level thread

the lock and access the critical region. Code implementation is shown in Fig. 6.

```
void lock_release(lock_t *l)
{

    l->held = 0;
}
```

**Figure 6:** Lock release for user level thread

## 7 FRISBEE TEST PROGRAM

We built a simple program that uses thread_create() to create a user defined number of threads. The threads will simulate a game of frisbee where each thread passes the frisbee (token) to the next. The location of the frisbee is updated in a critical region protected by our implemented user level lock lock. Each thread spins to check the value of the lock atomically. If it is its turn, then it prints a message, and releases the lock. We have provided a simulation of Frisbee game, the user specifies the number of threads (players) and the number of passes in the game. The parent process creates multiples threads (players), each thread accesses the lock, check if it is its turn to throw the frisbee (token), if so, it assigns the id of the next thread to receive the frisbee, increases the number of passes and releases the lock.

In Fig. 7, we can see the code of controlling the sequence of receiving frisbee. Initially we track the thread number. Total thread number will be given by the user while starting the program. Then it initializes the current pass number which will be printed in the prompt. After that we run a loop for total pass number times and check that current thread has not received the frisbee before. If it is true, then user level lock is acquired, pass number is increased, saves the ID that receives the frisbee in current iteration so that next it can not receive again and finally it releases the lock. Code implementation is shown in Fig. 7.

In Fig. 8, we can see that main function implementation of frisbee.c program. It at first acquires the lock and then receives the total thread count to be created from the user and also total round number for the frisbee game. Then a loop runs for total thread count times and its each iteration index is kept track and the function thread_create((void*)player,(void*)&arg) is called within it. play is the function for the game which is called by each thread after it is created by system call cloning() that we discussed before. After that it goes for sleep mode for some time and agian continues same task in the loop until the loop terminates. Then finally we notifies by promt if the game is over. We see total number of round and also who is passing the frisbee to which thread clearly. Detail impelementation is given in Fig. 8.

```
void *player(void *arg)
{
    int tid = *(uint64*)arg;
    int pass_num = pass_round;
    int i;
    for(i = 0;i < pass_num;i++)
    {
        if(thrower!=tid)
        {
            lock_acquire(&lock);
            pass++;
            printf("Pass number %d : ",pass);
            printf("Thread %d is passing the token to Thread %d\n",thrower,tid);
            thrower = tid;
            lock_release(&lock);
            sleep(20);
        }
        tid = (tid+1)%thread_num;
    }
    printf("Simulation of Frisbee game has finished, %d rounds were played in total!\n",pass_round);
    exit(0);
}
```

**Figure 7:** Code of simulating the frisbee game

```
int main(int argc, char *argv[])
{
    lock_init(&lock);
    thread_num = atoi(argv[1]);
    pass_round = atoi(argv[2]);

    int i;
    uint64 arg = 0;
    for(i=0;i<thread_num;i++)
    {
        arg = i+1;
        thread_create(player((void*)&arg),(void*)&arg);
        sleep(10);
    }
    sleep(40);
    exit(0);
}
```

**Figure 8:** Implementation of frisbee program

## 8 CONTRIBUTIONS OF EACH MEMBER.

The contributions of Md Abdullah Al Mamun are listed below.

- Lock initialization
- Lock acquire & release functions for spin lock
- Demo video recording

The contributions of Yicheng Zhang are listed below.

- Clone system call
- Thread create function
- Frisbee test program

## REFERENCES

[1] R. Cox, M. F. Kaashoek, and R. Morris. Xv6, a simple unix-like teaching operating system, 2011.