# Operating System CS 202

## Lab Assignment Report -III

**Group Member:** Madhusudhan Tungamitta(862325125), Akhilesh Reddy Gali(862258042), Srinivasa Biradavolu(862318458)

## Summary:

1. List Of Files Modified
2. Screenshots of the changes made
   a. clone() system call implementation in proc.c and exit(),wait() modifications.
   b. Thread create() library
3. Output screenshots
4. Running Commands
5. Demo Video Link

## 1. List of Files Modified:

We have created a new file “frisbee.c” ,“thread.c” and “thread.h” in the user folder and below are the files we made changes

| Folder Name | Files |
|---|---|
| Kernel | syscall.h |
| Kernel | syscall.c |
| Kernel | sysproc.c |
| Kernel | proc.c |
| Kernel | proc.h |
| Kernel | defs.h |
| user | usys.pl |
| user | user.h |
| user | thread.h |
| user | thread.c |
| user | frisbee.c |
| | MakeFile |

**Kernel\Syscall.h:**

By adding sys_clone in line 22 we are reserving a place for our own system call.

```
#define SYS_close  21
#define SYS_clone   22
```

**Kernel\Syscall.c**

We create an entry for the system call "SYS_clone" and its function "sys_clone" respectively in the system call table. Here we are adding that acts as a pointer to the system call where this file contains an array of function pointers to the system calls which are defined in different locations, So the sys_clone act as a pointer for our system call

```
extern uint64 sys_clone(void);
```

```
[SYS_close]   sys_close,
[SYS_clone]   sys_clone,
};
```

**Kernel\Sysproc.c**

As we created a new function sys_clone in the system call table we defined previously in the syscall.c file, we defined the sys_clone functions. The clear need for writing sysproc is that when the user wants to pass arguments to kernel-level functions from the user level function it is not directly possible in xv6.

```
uint64
sys_clone(void)
{
  uint64 var;
  int s;
  if(argaddr(0, &var) < 0)
  {
    return -1;
  }
  if(argint(1, &s) < 0)
    {
    return -1;
  }
```

**Kernel\trap.c:**

Trapframe provides the arguments to handle the trap and when the cpu changes from user mode to kernel, trapframe saves the userspace registers.

```
uint64 fn = TRAMPOLINE + (userret - trampoline);
((void (*)(uint64,uint64))fn)(TRAPFRAME - (4096 * p->threadid), satp);
```

**Kernel\Proc.c:**

Here we have added the next threadid value and spinlock which is an alternative for blocking synchronization.

```
int nextpid = 1;
int nextthreadid = 1; // Lab3

struct spinlock pid_lock;
struct spinlock threadid_lock; //Lab3
```

Allocation thread id to the procid() function which will increase the id by one and return the id value

```
found:
  p->pid = allocpid();
  p->state = USED;
  p->threadid = 0;
```

```
found:
  p->pid = allocpid();
  p->state = USED;
  p->threadid = allocpid();
```

Clone system call is used to implement the threads, 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 actual. In the Clone function if the stack is equal to the zero then it returns nothing if it is not empty then allocate the thread to the value np and mappages which takes a take a pagetable directory, page size and modifies the page directory.we copy the saved user registers to trapframe->sp and assign the a0 value to zero because fork the child to zero

**Address space:**

```
//Took the reference From the above Fork function for implementing the clone function Because we are using parents address space and we are not creating a new address space
```

**Stack arguments:**

```
// stack acts as a starting point and it acts as a sanity check whether the stack is null or not Because parent allocates the stack area before the call is mode
```

```c
//Took the reference From the above Fork function
int
clone(void *stack)
{
  int size = 4096 * sizeof(void);
  int i, threadid;
  struct proc *np;
  struct proc *p = myproc();
  if (stack == 0) {
    return -1;
  }
  if((np = allocproc_thread()) == 0){   // Allocate thread.
    return -1;
  }
  np->pagetable = p->pagetable;
  if(mappages(np->pagetable, TRAPFRAME - (4096 * np->threadid), 4096,
              (uint64)(np->trapframe), PTE_R | PTE_W) < 0){
    uvmunmap(np->pagetable, TRAMPOLINE, 1, 0);
    uvmfree(np->pagetable, 0);
    return 0;
  }
  np->sz = p->sz;
  *(np->trapframe) = *(p->trapframe);
  np->trapframe->sp = (uint64) (stack + size);
  np->trapframe->a0 = 0;
  for(i = 0; i < NOFILE; i++)// increment reference counts on open file descriptors
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);
  safestrcpy(np->name, p->name, sizeof(p->name));
```

**Wait() Modification:**

```
// Some resources are now shared by all the threads of some process in
order to avoid we deallocate the thread local resources to the n->parent
```

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

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

  return threadid;
```

**Changes in exit() function:**

It helps in closing the files and reparent the thread id, executes only if it is parent

```
if(p->threadid == 0)
reparent(p);//remoditfication for Lab3
```

**Kernel\proc.h:**

```
  char name[16];               // Process name (debugging)
  int threadid;
};
```

**Kernel\defs.h**

```
int             clone(void*,int); // Forward de
```

**User\user.h:**

```
int clone(void*,int); // Lab3
```

**User\usys.pl**

This is the entry point for the info

```
entry("uptime");
entry("clone"); #Clone sys call for user
```

**Make File:**

Changing the lines 72 and 73 for controlling the command line

```
$U/_wc\
$U/_zombie\
$U/_frisbee\
```

Here we make the entry file for the frisbee file

```
ULIB = $U/ulib.o $U/usys.o $U/printf.o $U/umalloc.o $U/thread.o
```

Adding the thread.o will helps the thread will compile as library

**User\thread.h:**

This header file contains all the macros and function declarations for thread.c

```
struct  lock_t
{
    uint unlocked;
};
typedef struct lock_t lock_t;
int thread_create(void *(*thread_fn)(void*), void *arg);
void lock_init(lock_t *lock);
void lock_acquire(lock_t *);
void lock_release(lock_t *);
```

## User\thread.c:

We implement simple user-level spin lock routines: lock_acquire() and lock_release(), which acquire and release the lock and lock_init() which is used to initialize the lock.

```
#include "kernel/spinlock.h"

int thread_create(void *(*thread_fn)(void*), void *arg) {
    int threadid;
    void* stack = (void*)malloc(4096 * sizeof(void));
    threadid  = clone(stack,4096 * sizeof(void));
    if(threadid != 0) {
    }
    else{
    (*thread_fn) (arg);
    exit(0);
    }
    return 0;
}

//Lock implementation
void lock_init(lock_t *lock)
{
    lock->unlocked = 0;
}

void lock_acquire(lock_t *lock)
{
    while(__sync_lock_test_and_set(&lock->unlocked, 1) != 0);
  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen strictly after the lock is acquired.
  // On RISC-V, this emits a fence instruction.
    __sync_synchronize();
}
void lock_release(lock_t *lock)
{
  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen strictly after the lock is acquired.
  // On RISC-V, this emits a fence instruction.
    __sync_synchronize();
      // Release the lock, equivalent to lk->locked = 0.
  // This code doesn't use a C assignment, since the C standard
  // implies that an assignment might be implemented with
```

```
  // implies that an assignment might be implemented with
  // multiple store instructions.
  // On RISC-V, sync_lock_release turns into an atomic swap:
  //   s1 = &lk->locked
  //   amoswap.w zero, zero, (s1)
    __sync_lock_release(&lock->unlocked,0);
}
```

## User\frisbee.c:

This is the user program that uses thread_create() to create threads, when we run the user program in qemu on the terminal the threads will simulate a game of frisbee or token to the next thread. This code is already provided in the instructions and is used as it is.

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

lock_t lock;
int n_threads, n_passes, cur_turn, cur_pass;

void* thread_fn(void *arg)
{
  int thread_id = (uint64)arg;
  int done = 0;
  while (!done) {
    lock_acquire(&lock);
    if (cur_pass >= n_passes) done = 1;
    else if (cur_turn == thread_id) {
      cur_turn = (cur_turn + 1) % n_threads;
      printf("Round %d: thread %d is passing the token to thread %d\n",
        ++cur_pass, thread_id, cur_turn);
    }
    lock_release(&lock);
    sleep(0);
  }
  return 0;
}

int main(int argc, char *argv[])
{
  if (argc < 3) {
    printf("Usage: %s [N_PASSES] [N_THREADS]\n", argv[0]);
    exit(-1);
  }
  n_passes = atoi(argv[1]);
  n_threads = atoi(argv[2]);
  cur_turn = 0;
  cur_pass = 0;
  lock_init(&lock);
  for (int i = 0; i < n_threads; i++) {
    thread_create(thread_fn, (void*)(uint64)i);
  }
  for (int i = 0; i < n_threads; i++) {
    wait(0);
  }
  printf("Frisbee simulation has finished, %d rounds played in total\n", n_passes);

  exit(0);
}
```

## Output Screenshot

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ frisbee 6 4
Round 1: thread 0 is passing the token to thread 1
Round 2: thread 1 is passing the token to thread 2
Round 3: thread 2 is passing the token to thread 3
Round 4: thread 3 is passing the token to thread 0
Round 5: thread 0 is passing the token to thread 1
Round 6: thread 1 is passing the token to thread 2
Frisbee simulation has finished, 6 rounds played in total
$ frisbee 20 7
Round 1: thread 0 is passing the token to thread 1
Round 2: thread 1 is passing the token to thread 2
Round 3: thread 2 is passing the token to thread 3
Round 4: thread 3 is passing the token to thread 4
Round 5: thread 4 is passing the token to thread 5
Round 6: thread 5 is passing the token to thread 6
Round 7: thread 6 is passing the token to thread 0
Round 8: thread 0 is passing the token to thread 1
Round 9: thread 1 is passing the token to thread 2
Round 10: thread 2 is passing the token to thread 3
Round 11: thread 3 is passing the token to thread 4
Round 12: thread 4 is passing the token to thread 5
Round 13: thread 5 is passing the token to thread 6
Round 14: thread 6 is passing the token to thread 0
Round 15: thread 0 is passing the token to thread 1
Round 16: thread 1 is passing the token to thread 2
Round 17: thread 2 is passing the token to thread 3
Round 18: thread 3 is passing the token to thread 4
Round 19: thread 4 is passing the token to thread 5
Round 20: thread 5 is passing the token to thread 6
Frisbee simulation has finished, 20 rounds played in total
$
```

```
$ frisbee 20 20
Round 1: thread 0 is passing the token to thread 1
Round 2: thread 1 is passing the token to thread 2
Round 3: thread 2 is passing the token to thread 3
Round 4: thread 3 is passing the token to thread 4
Round 5: thread 4 is passing the token to thread 5
Round 6: thread 5 is passing the token to thread 6
Round 7: thread 6 is passing the token to thread 7
Round 8: thread 7 is passing the token to thread 8
Round 9: thread 8 is passing the token to thread 9
Round 10: thread 9 is passing the token to thread 10
Round 11: thread 10 is passing the token to thread 11
Round 12: thread 11 is passing the token to thread 12
Round 13: thread 12 is passing the token to thread 13
Round 14: thread 13 is passing the token to thread 14
Round 15: thread 14 is passing the token to thread 15
Round 16: thread 15 is passing the token to thread 16
Round 17: thread 16 is passing the token to thread 17
Round 18: thread 17 is passing the token to thread 18
Round 19: thread 18 is passing the token to thread 19
Round 20: thread 19 is passing the token to thread 0
Frisbee simulation has finished, 20 rounds played in total
$
```

## 4. Running Commands:

**Running on Windows subsystem:**

1. Install ubuntu from Microsoft store
2. Run the below commands

**$ sudo apt-get update && sudo apt-get upgrade**

**$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu**

3. Copy our project and got to the folder “cd xv6-riscv”

**Running Code:**

4. Run the command “make qemu”
    a. Check with the following command “frisbee 6 4”

## 5.Demo Link:

https://drive.google.com/file/d/1COIqpi_hCAaaVCqWnNeIF5Hx_NkSpYlm/view?usp=sharing

CS 202 - Lab Assignment -III
Spring 2022