CS344 : ASSIGNMENT-1

C40 Group Members :-

1. Abhinav Choudhary (200101005)
2. Yash Garg (200101113)
3. Nayanika Ghosh (200123036)
4. Aarti Meena (180101001)

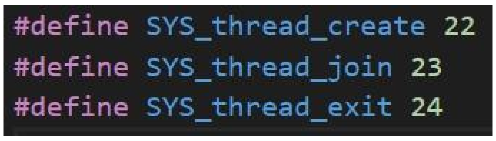
Part 1 :- Kernel threads

Here, we implemented the kernel threads from scratch.
We defined three system calls,namely :-
i) 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. 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 is one page in size. The new thread starts executing at the address specified by **fcn**.

ii) int thread_join(void) :- This call waits for a child thread that shares the address space with the calling process. It returns the PID of the waited-for child or -1 if none.

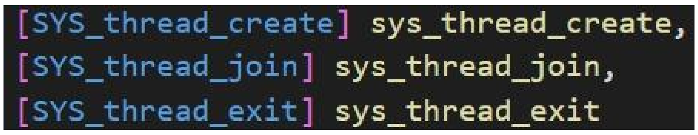
iii) int thread_exit(void) :- This call allows a thread to terminate.

We began the procedure by editing **syscall.h** in which a specific number is assigned to each system call. To add our custom system call, we added the following lines to the file.

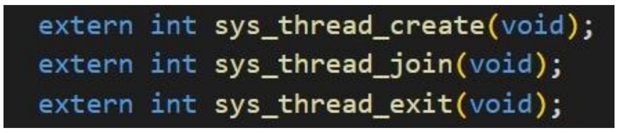


```
#define SYS_thread_create 22
#define SYS_thread_join 23
#define SYS_thread_exit 24
```

Then we added a pointer to the system call in **syscall.c** file in the **xv6-public** directory.To add our custom system call, we added the following lines in two different locations of the file.

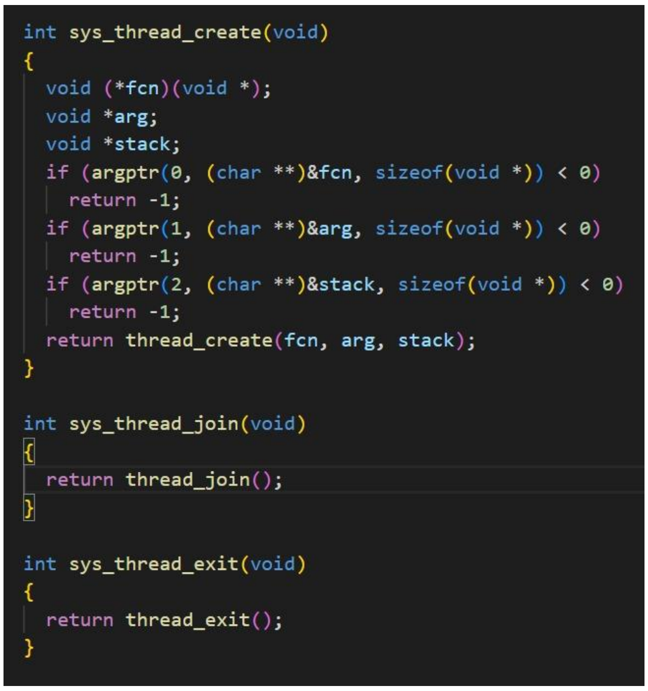


```
[SYS_thread_create] sys_thread_create,
[SYS_thread_join] sys_thread_join,
[SYS_thread_exit] sys_thread_exit
```



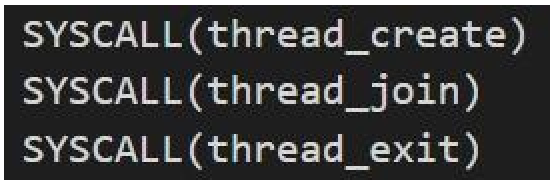
```
extern int sys_thread_create(void);
extern int sys_thread_join(void);
extern int sys_thread_exit(void);
```

We added three functions in **sysproc.c** file,namely,
i) int sys_thread_create(void) :- it calls **thread_create** function which we implement later in **proc.c** file.
ii) int sys_thread_join(void) :- it calls **thread_join** function which we implement later in **proc.c** file.
iii) int sys_thread_exit(void) :- it calls **thread_exit** function which we implement later in **proc.c** file.



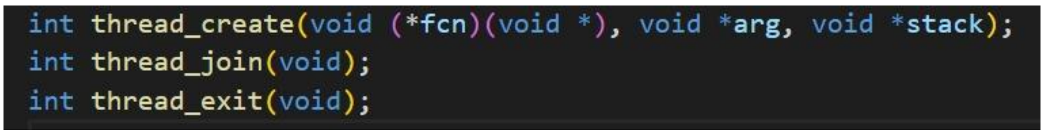
```
int sys_thread_create(void)
{
  void (*fcn)(void *);
  void *arg;
  void *stack;
  if (argptr(0, (char **)&fcn, sizeof(void *)) < 0)
    return -1;
  if (argptr(1, (char **)&arg, sizeof(void *)) < 0)
    return -1;
  if (argptr(2, (char **)&stack, sizeof(void *)) < 0)
    return -1;
  return thread_create(fcn, arg, stack);
}

int sys_thread_join(void)
{
  return thread_join();
}

int sys_thread_exit(void)
{
  return thread_exit();
}
```

Now, in order to add an interface for a user program to call the system call, we added the following lines to **usys.S** file,



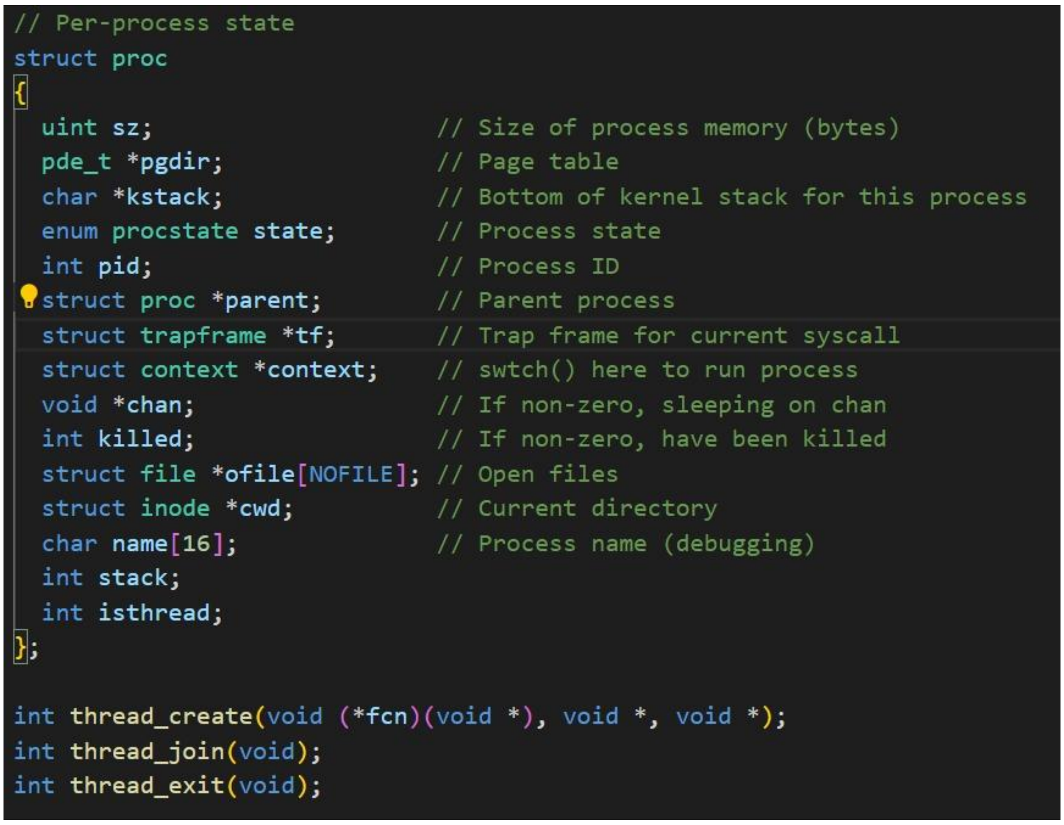
```
SYSCALL(thread_create)
SYSCALL(thread_join)
SYSCALL(thread_exit)
```

and the following lines to **user.h** file.



```
int thread_create(void (*fcn)(void *), void *arg, void *stack);
int thread_join(void);
int thread_exit(void);
```

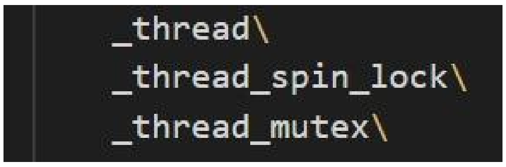
Then we modified the **proc.h** file. We added two int variables, **stack** and **isthread** to the structure **proc**. We also added the prototypes of the three functions **thread_create**, **thread_join** and **thread_exit** to the file.



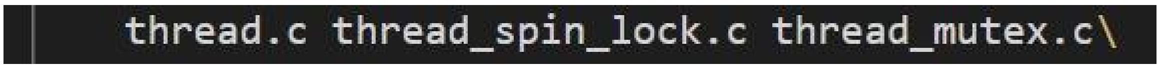
```
// Per-process state
struct proc
{
  uint sz;                     // Size of process memory (bytes)
  pde_t *pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int stack;
  int isthread;
};

int thread_create(void (*fcn)(void *), void *, void *);
int thread_join(void);
int thread_exit(void);
```

We implemented the three system call functions in **proc.c** file. The implementations can be seen in the file we have attached.

Finally, we modified the **Makefile** in two different locations to add the newly created user programs and test the creation of the threads and the execution of the codes.

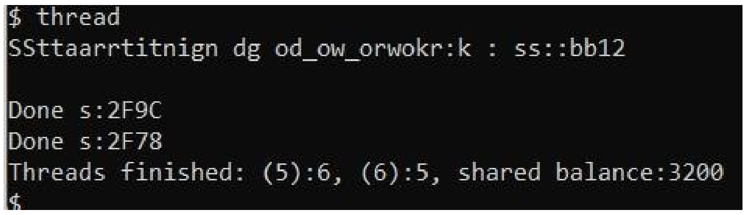


```
	_thread\
	_thread_spin_lock\
	_thread_mutex\
```



```
	thread.c thread_spin_lock.c thread_mutex.c\
```

Now, we run the commands **"make clean"**, **"make"** and **"make qemu-nox"** in the **xv6-public** directory. Then we run **thread.c** in the terminal of the **xv6-public** directory. The following is the output :-



```
$ thread
SSttaarrtitnign dg od_ow_orwokr:k : ss::bb12

Done s:2F9C
Done s:2F78
Threads finished: (5):6, (6):5, shared balance:3200
$
```

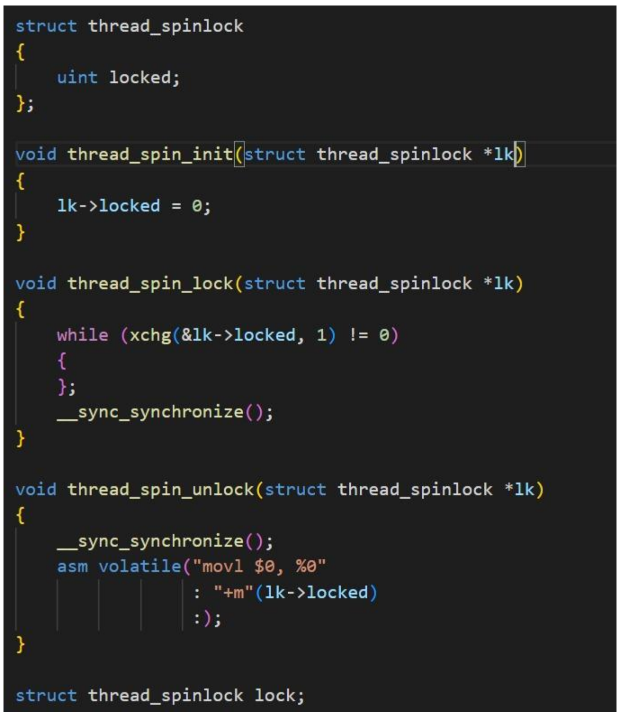
The second line of the screenshot comes because of context switching among the threads as we have not implemented any lock. So, the printf is context switched in between, hence a shuffled string of outputs of both the processes is printed.
Also, since both the processes used shared memory in the form of balance and no lock is implemented, the value of the **shared balance** is not equal to **6000**, but **3200** which is not the correct value.

Part 2 :- Synchronization

In this part, we implemented synchronization mechanism using spinlocks and mutexes.

Spinlock :-

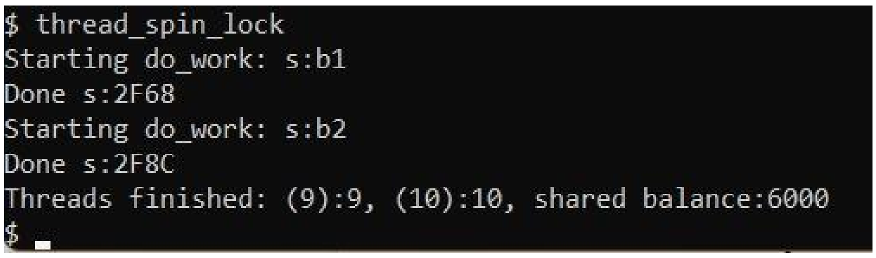
Spinlock is a locking system mechanism. A thread has to wait in a loop until a lock is available. A single thread is permitted at a time to acquire a lock and continue in the critical section. Spinlock can be used only for mutual exclusion.



```
struct thread_spinlock
{
    uint locked;
};

void thread_spin_init(struct thread_spinlock *lk)
{
    lk->locked = 0;
}

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

void thread_spin_unlock(struct thread_spinlock *lk)
{
    __sync_synchronize();
    asm volatile("movl $0, %0"
                 : "+m"(lk->locked)
                 :);
}

struct thread_spinlock lock;
```

The above screenshot shows the portion of the code that we have added from ourselves to the newly created **thread_spin_lock.c** file.
The structure **thread_spinlock** contains a **uint** variable **locked** which is used to check if the lock is held or not.
The function **thread_spin_init** is used to initialise the **locked** variable to 0.
The function **thread_spin_lock** is used to acquire the lock. It spins until the lock is acquired.
The function **thread_spin_unlock** is used to release the lock.

We added the **thread_spin_lock.c** to the **Makefile** (screenshot attached previously).
Now, we run the commands **"make clean"**, **"make"** and **"make qemu-nox"** in the **xv6-public** directory.
The output after the execution of the program **thread_spin_lock.c** is as follows :-

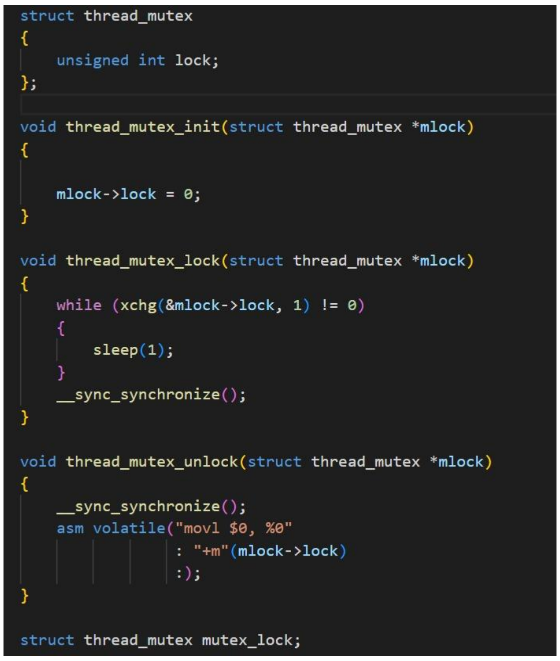


```
$ thread_spin_lock
Starting do_work: s:b1
Done s:2F68
Starting do_work: s:b2
Done s:2F8C
Threads finished: (9):9, (10):10, shared balance:6000
$ 
```

As observed, the value of the **shared balance** comes out to be **6000** which is the expected value.

Mutex :-

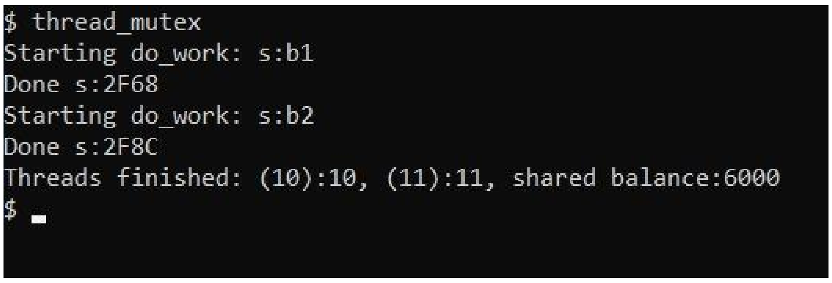
Mutex is also a locking mechanism. A thread acquires the mutex before accessing the shared resource. The thread releases the lock later. In contrast to spinlock, mutex allows the threads to take turns sharing the same resources. The task can sleep while waiting for the lock in mutex, unlike that in spinlock.



```
struct thread_mutex
{
    unsigned int lock;
};

void thread_mutex_init(struct thread_mutex *mlock)
{

    mlock->lock = 0;
}

void thread_mutex_lock(struct thread_mutex *mlock)
{
    while (xchg(&mlock->lock, 1) != 0)
    {
        sleep(1);
    }
    __sync_synchronize();
}

void thread_mutex_unlock(struct thread_mutex *mlock)
{
    __sync_synchronize();
    asm volatile("movl $0, %0"
                 : "+m"(mlock->lock)
                 :);
}

struct thread_mutex mutex_lock;
```

The above screenshot shows the portion of the code that we have added from ourselves to the newly created **thread_mutex.c** file.
The structure **thread_mutex** contains an **unsigned int** variable **locked** which is used to check if the mutex lock is held or not.
The function **thread_mutex_init** is used to initialise the **locked** variable to 0.
The function **thread_mutex_lock** is used to acquire the lock. It calls the **sleep** function with 1 as the parameter and it works similar to the **yield** function.
The function **thread_mutex_unlock** is used to release the lock.

We added the **thread_mutex.c** to the **Makefile** (screenshot attached previously). Now, we run the commands **"make clean"**, **"make"** and **"make qemu-nox"** in the **xv6-public** directory.
The output after the execution of the program **thread_mutex.c** is as follows :-



```
$ thread_mutex
Starting do_work: s:b1
Done s:2F68
Starting do_work: s:b2
Done s:2F8C
Threads finished: (10):10, (11):11, shared balance:6000
$ 
```

As observed, the value of the **shared balance** comes out to be **6000** which is the expected value.