ASSIGNMENT - 1

Group Number : C19
Group members : Bokka Srividya - 200101026
Yenugonda Suma Sree - 200101114
Jeram Niharika - 200101046

Part 1: Kernel Threads

Aim : To implement kernel threads by adding system calls , specifically thread_create , thread_join , thread_exit without any process synchronization methods.

To create the system calls required , following files were edited :

syscall.h : Here we assign a unique number to every system call in the xv6 system. Before adding the new syscalls there were already 22 system calls
Now we add
- #define SYS_thread_create 23
- #define SYS_thread_join 24
- #define SYS_thread_exit 25

syscall.c : It stores an array of pointers to functions (syscalls[]) which use indexes defined in syscall.h to point to a respective system call function stored at a different memory location.
- extern int sys_thread_create(void);
- extern int sys_thread_join(void);
- extern int sys_thread_exit(void);

We also put a function prototype here(but not implementation).
- [SYS_thread_create] sys_thread_create,
- [SYS_thread_create] sys_thread_join,
- [SYS_thread_create] sys_thread_exit.

sysproc.c: Here the implementations of system calls are written.

user.h and **usys.S**: These are the interfaces for the system to access the system calls. The function prototype is added in user.h (included as header file in our program) while instruction to treat it as a system call is included in usys.S

user.h :

- int thread_create(void()(void), void*, void*);
- int thread_join(void);
- void thread_exit(void);

usys.S :

- SYSCALL(thread_create)
- SYSCALL(thread_join)
- SYSCALL(thread_exit)

In **sysproc.c** the implementation of system calls we just called the function .

thread_create, **thread_join** and **thread_exit** are implemented in **proc.c** and declared in **defs.h**.

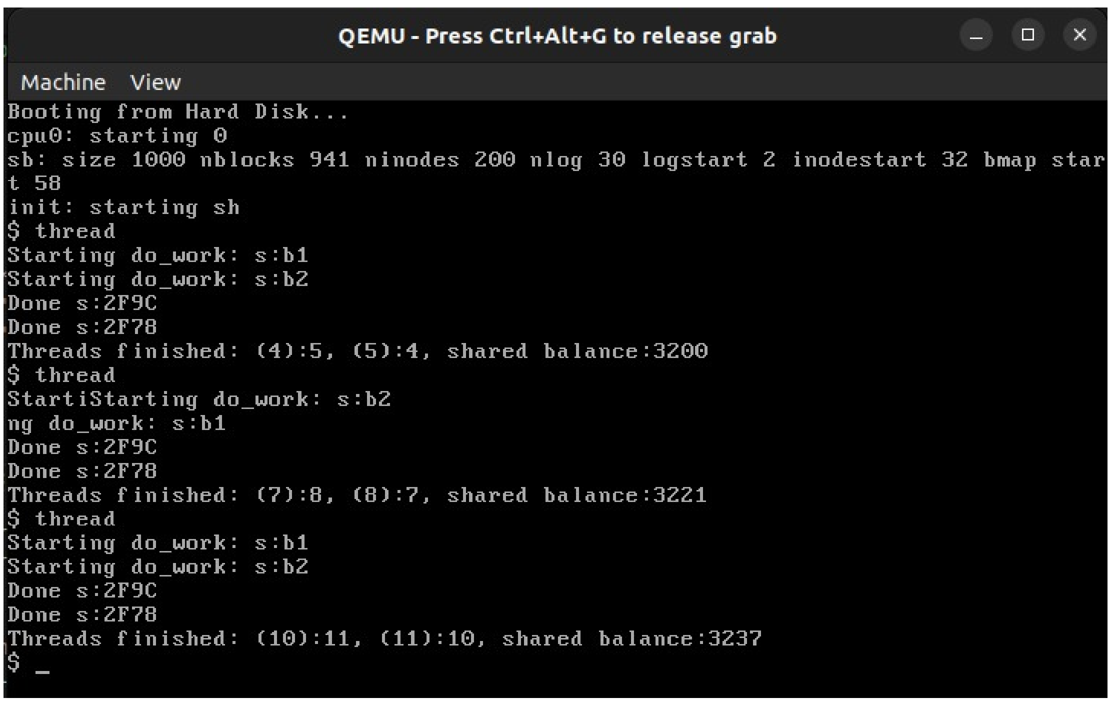
thread_create():
We implemented thread_create similar to **fork()**, but in fork() the allocation of address space to the child process is different from the parent process and in our thread_create we gave the **same address** for the new process.
myproc() returns the currently running process in the cpu.
*struct proc *curproc = myproc();*
Initially we create a new process we give it a *pid* for it and change the state of the process to **EMBRYO**.
np->pgdir = curproc->pgdir;
np->tf->eip = (uint)fcn;
We then set the instruction pointer to starting point of the function address
np->tf->esp = (uint)stack+4096;
And then we go to the end of the stack
Stack pointer go back by 4 bytes and place the arg at that address.
Other code in the thread_create is same as fork like duplicating the open file to the thread,current working directory and name of the process
After that it is changing the process state **EMBRYO** to **RUNNABLE**

thread_join():
thread_join system call is used to make a process to wait while the other is in execution in cpu till its completion
thread_join returns the exited thread process id.
Initially it goes through the process table and searches for the children of the main process (which are threads here) and checks them if they are exited or not . If we have children and any of them are not exited then it makes the main process to wait(sleep) until one thread terminates.

And we are not freeing the page directory (because main process also sharing same address space)
If thread is terminated then it reinitializes the process to an unused process and returns the pid.

thread_exit():
Fetches the currently running thread and close it all files and wakeup the parent process. If it has children processes then it will change the parent of that childrens to initproc. If any of that children is already terminated then it will wakeup the initproc set the state of the process to the ZOMBIE

Observations :
Since there is no synchronization between the threads they may run into race conditions which are caused due to the shared variable (here the shared variable is “shared balance”) simultaneously .



i) Here when initially threads were generated the balance of thread1 was 3200 and that of thread2 was 2800
ii) The total shared balance to be shown was 6000 .
iii) But we observe that the shared balance shown is less than 6000
iv) This is because the threads run into race condition due to the shared variable – “shared balance”.

Part 2 : Synchronization

Aim : To fix the synchronization error using spinlock and mutex primitives so that the threads would execute atomically

Description :

1)Spinlock:

We implemented spinlocks in the file thread.c in which we defined a thread_spinlock data structure that has the attribute “**locked”**, which indicates when the lock is acquired or not.

i) void thread_spin_init(struct thread_spinlock *lk):This function initializes the lock to the correct initial state.We initialized locked attribute in thread_spinlock to zero.When “locked” is equal to zero states that the thread_spinlock is not acquired.

ii) void thread_spin_lock(struct thread_spinlock *lk):

In this, our implementation is such that we acquire the lock after using an atomic instruction xchg(Exchange).If the returned value of this xchg function is not equal to zero we enter into busy waiting. Else we use the atomic instruction “__sync_synchronize()” that tells 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 after the lock is acquired.

iii) void thread_spin_unlock(struct thread_spinlock *lk):

In this we use “__sync_synchronize()” and then we release the lock, equivalent to setting locked=0. We used asm volatile to set this locked value to zero because this code can’t use a C assignment as it might not be atomic.

2)Mutexes:

Mutexes can be implemented similar to spinlocks except that it keeps the other processes or threads which are not in execution to sleep rather than busily waiting.

i) void thread_mutex_init(struct thread_mutexlock *lk):

This function initializes the lock to the correct initial state.We initialized locked attribute in thread_mutexlock to zero.

When “locked” is equal to zero states that the thread_mutexlock is not acquired.

ii) void thread_mutex_lock(struct thread_mutexlock *lk):

In this, our implementation is such that we acquire the lock after using an atomic instruction xchg(Exchange).If the returned value of this xchg function is not equal to zero that is when the lock is already acquired, the thread can sleep while waiting for the lock to be released . Else we use the atomic instruction “__sync_synchronize()” that tells 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 after the lock is acquired.

iii) void thread_mutex_unlock(struct thread_mutexlock *lk):

In this we use “__sync_synchronize()” and then we release the lock, equivalent to setting locked=0. We used asm volatile to set this locked value to zero because this code can’t use a C assignment as it might not be atomic.

Observations : Since we used the synchronization primitives the shared balance is used by only one thread which locks the shared variable , until the thread is executed and releases the lock the other thread cannot access the shared variable .

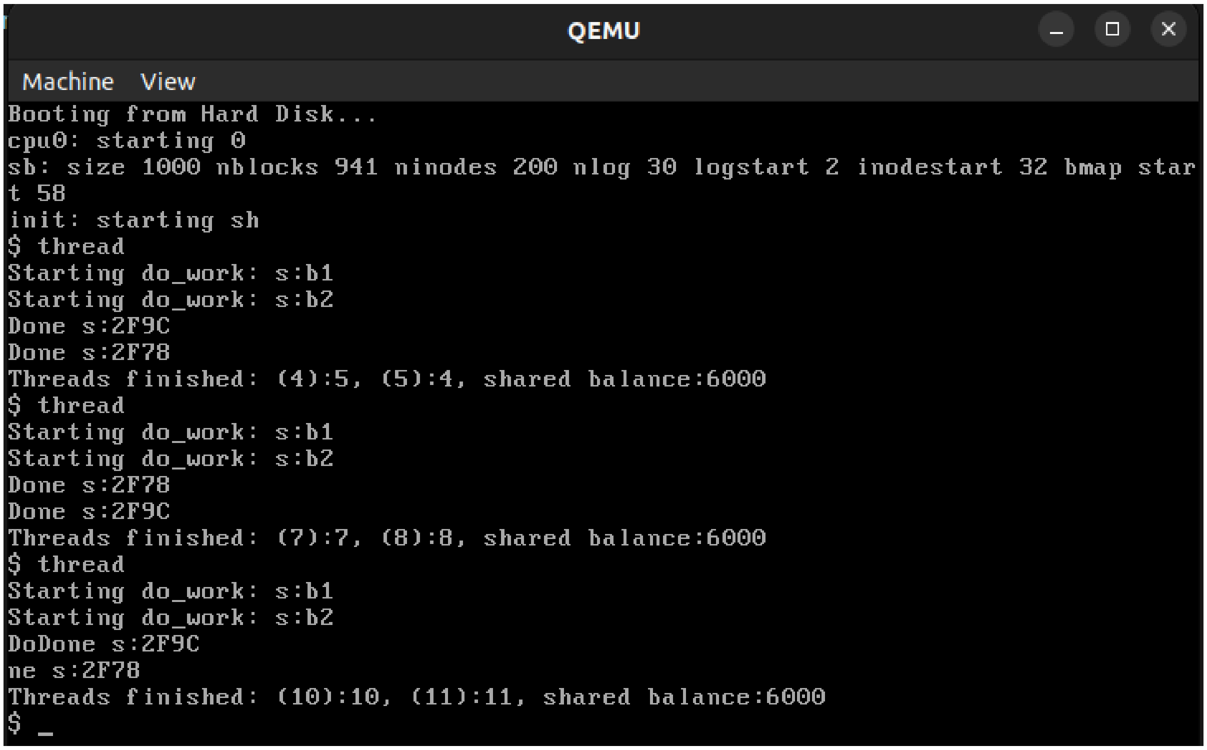


Fig 01 : Threads execution with spinlocks

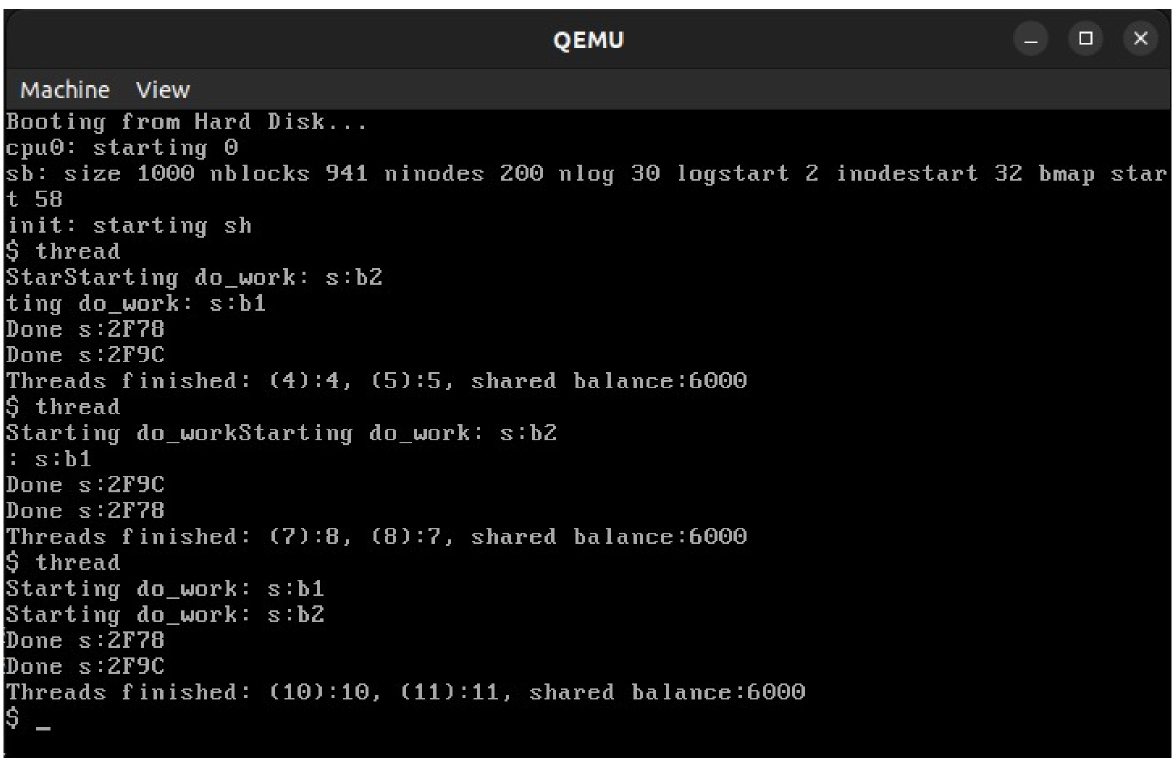


Fig 02 : Threads execution with mutexes