

MP6 – Operating Systems

Garuda Suma Pranavi – UIN: 926009146

Improvements in MP5 after TA discussion:

I was asked to make threads 3 and 4 terminating in MP5. When we terminate all the threads (fun 3 and fun 4 also), we get an exception error 6 as we don't have an idle thread (like a real system which is always available, so that we can transfer control to it). In our implementation, we just cannot pass_on anywhere leading to this error. Thus, we must keep at least Thread 4 non-terminating to avoid the error.

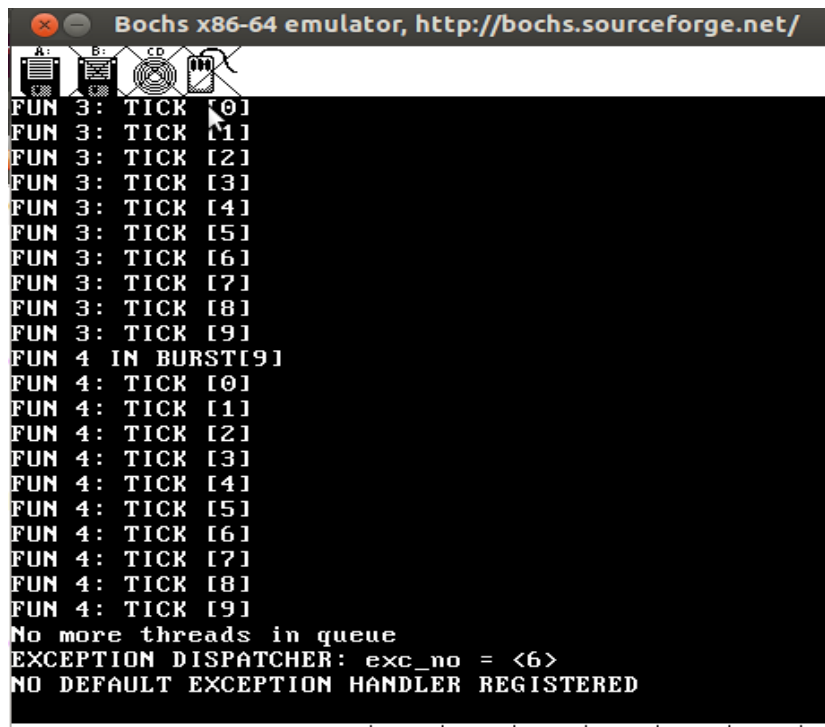
Changes made to code: thread.C

```
static void thread_shutdown() {
    /* This function should be called when the thread returns from
    thread function.
    */

    Machine::disable_interrupts();
    SYSTEM_SCHEDULER -> terminate(current_thread);

    delete current_thread;

    if (SYSTEM_SCHEDULER -> tail != NULL)
        {SYSTEM_SCHEDULER -> yield();}
    else
        {Console::puts("No more threads in queue\n");}
```



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 4 IN BURST[9]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
No more threads in queue
EXCEPTION DISPATCHER: exc_no = <6>
NO DEFAULT EXCEPTION HANDLER REGISTERED
```

MP6:

I have implemented a Blocking Disk function which adds a layer on top of the SimpleDisk functions to support the same read and write operations without busy waiting (i.e. without polling) in the device driver code.

In my implementation, `wait_until_ready()` adds the current thread to the queue and yield the CPU until the operation is complete. This was easy to do because of the linked list approach I used in making my scheduler in MP5.

I don't think we are able to see any delays in read/write due to inaccurate emulation in Bochs discussed in the assignment. However, I think my implementation would work in a real world scenario.

I have also included bonus options 1 and 3 and made appropriate changes in `kernel.C` to reflect the same.

Bonus 1:

I have implemented the support for Disk Mirroring by using MASTER and SLAVE controller disk ID's. We read data from both master and slave disks (by passing their IDs in the issue operation function). I'm not really sure how I can assign one thread to the master and one to the slave to check which one returns fast. The write function also writes to both the master and slave disks. I've also made changes to the `kernel.C` to check the working of the mirrored disk.

Bonus 3:

Design of a thread safe disk system – We can use a LazyList to design the concurrent data structure (i.e. for a disk to be accessed by multiple threads concurrently). However simpler ways of doing option 4 can consider implementing a mutex. The mutex acting as a mutually exclusive flag allows only one thread to access a section of code blocking access to others. Thus, implementing the design requirement specified in Option3 that a disk is to be accessed at most one thread at a time. For threads this can be done through `pthread_mutex_lock()` and `pthread_mutex_unlock()` once the action is complete. `Lock()` will be called when a `read()` or `write()` operation is called and then `unlock()` once the `read()/write()` is over. However, if another thread wants to read/write, it has to wait for the thread to be unlocked.

I have just mentioned the design, not implemented Bonus 4.