

MP5: Kernel-Level Thread Scheduling

Sushant Vijay Shelar
UIN: 733001479
CSCE611: Operating System

Assigned Tasks

Main: Completed.

Bonus Option 1: Completed

System Design

In this machine problem, we added scheduling of multiple kernel-level threads.

1. **User Process:** This box represents the user-level processes that create threads.
2. **Thread Scheduler:** The core component within the kernel that schedules threads according to the FIFO algorithm.
3. **FIFO Queue:** Depicted as a vertical line or a series of slots, this shows where threads are lined up in the order they arrive.
4. **CPU Execution:** This is where the threads are actually executed. There might be one or more boxes representing CPU cores.

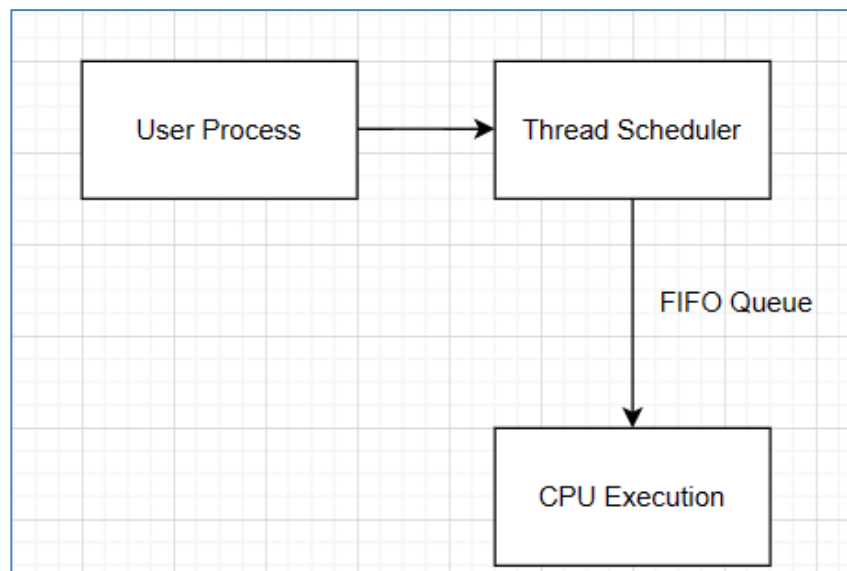


Figure 1: High Level FIFO Thread Scheduling System Design Diagram

Code Description

Code changes are made in Scheduler.C, Scheduler.H, Thread.C, Kernel.C files.

FIFO Kernel scheduler:

In the First-In-First-Out (FIFO) scheduling approach, threads are ordered and processed according to their arrival sequence. To facilitate this, a structure known as 'threadQueue' has been utilized, which is essentially a linked list that maintains the sequence of threads. The choice of a linked list is due to its efficient removal operations. Two pointers, labeled 'head' and 'tail,' are designated to mark the beginning and end of the queue, respectively. Several operations have been defined: one to insert threads into the queue, another to remove a specific thread from the queue, and a third to retrieve the next thread in sequence for processing. An illustrative snapshot that captures this setup is provided below,

```
struct threadQueue
{
    Thread *thread;
    threadQueue * next;
    static threadQueue* Head;
    static threadQueue* Tail;
    void enqueueThread(Thread* t);
    Thread* dequeueThread();
    void removeThread(Thread * t);
    bool isempty();
    threadQueue()
    {
        thread= nullptr;
        next=nullptr;
    }
};
```

Figure 2: threadQueue

EnqueueThread:

It adds a particular thread at the end of the ready queue.

```
void threadQueue::enqueueThread(Thread* t) {
    auto newElement = new threadQueue(); // Using 'auto' for type inference
    newElement->thread = t;

    if (Tail != nullptr) {
        Tail->next = newElement;
        Tail = newElement; // Directly updating 'Tail' without using 'Tail->next'
    } else {
        Head = Tail = newElement; // Merging two lines into one for the 'else' case
    }
}
```

Figure 3: EnqueueThread

dequeueThread:

Dequeue the current queue and return the thread.

```
Thread* threadQueue::dequeueThread() {
    // Check if the queue is empty before attempting to remove an item
    if (Head == nullptr) {
        return nullptr;
    }

    // Retrieve the thread from the head of the queue
    Thread* next_thread = Head->thread;
    // Save the next node to update the head after removal
    threadQueue* next_node = Head->next;

    // Delete the old head of the queue
    delete Head;
    // Update the head to the next node
    Head = next_node;

    // If after removal, the head is null, make sure the tail is also updated
    if (Head == nullptr) {
        Tail = nullptr;
    }

    // Return the thread from the removed queue node
    return next_thread;
}
```

Figure 3: dequeueThread

Yield:

- The yield method within the Scheduler class provides a mechanism for the current thread to step aside for others in the queue. If no threads are queued, it simply terminates.
- When other threads are pending, it secures the operation by turning off interrupts, chooses the next thread to run, and then hands over execution. After the switch, it reinstates the initial interrupt settings to maintain system stability.
- For bonus 1, interrupts are disabled at the entry of the function and enabled just before exiting the function.

```
void Scheduler::yield() {
    // assert(false);
    // Return immediately if there are no threads ready to run
    if (ready.isEmpty()) {
        return;
    }

    // Disable interrupts if they are currently enabled
    bool wereInterruptsEnabled = interruptIsOn();
    if (wereInterruptsEnabled) {
        setInterrupt(false);
    }

    // Remove the next thread from the ready queue and dispatch to it
    Thread* t = ready.dequeueThread();
    Thread::dispatch_to(t);

    // Restore the previous interrupt state
    if (wereInterruptsEnabled) {
        setInterrupt(true);
    }
}
```

Figure 4: Yield

Scheduler:

Assigns thread and next pointers as null pointer.

Resume:

- The resume method in the Scheduler class ensures a secure enlistment of a thread into the ready queue. Initially, it confirms whether interrupts are active and deactivates them to avert concurrent execution issues.
- Subsequently, the thread is queued for execution, and interrupts are restored to their initial state, guaranteeing that the thread is added without disrupting the system's operation.

```
void Scheduler::resume(Thread * _thread) {  
    // assert(false);  
    if(interruptIsOn())  
    {  
        setInterrupt(false);  
    }  
    ready.enqueueThread(_thread);  
    if(!interruptIsOn())  
    {  
        setInterrupt(true);  
    }  
}
```

Figure 5: Resume

Add:

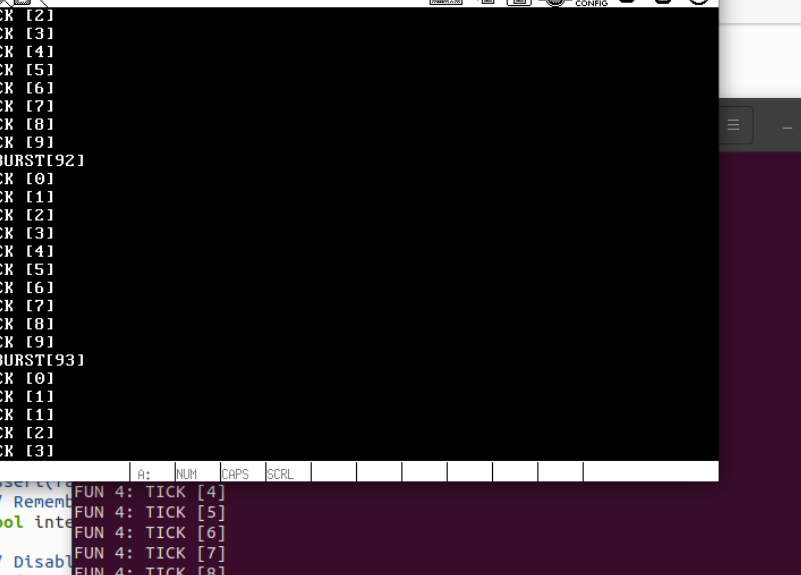
- The add function of the Scheduler class inserts a new thread into the queue, checking and handling interrupts to ensure safe operation.
- It deactivates interrupts if needed, adds the thread, then reverts interrupt settings, preserving the system's responsive and secure state.

```
void Scheduler::add(Thread* _thread) {  
    // assert(false);  
    // Remember the original state of interrupts  
    bool interruptsOriginallyEnabled = interruptIsOn();  
  
    // Disable interrupts if they were originally enabled  
    if (interruptsOriginallyEnabled) {  
        setInterrupt(false);  
    }  
  
    // Add the thread to the ready queue  
    ready.enqueueThread(_thread);  
  
    // Re-enable interrupts only if they were enabled originally  
    if (interruptsOriginallyEnabled) {  
        setInterrupt(true);  
    }  
}
```

Figure 6: add method

Testing

I used the 'make' command to compile all the files in the "mp5" folder. Then, I ran a script called 'copy kernel' to copy the bin file to the disk. After that, I executed the code using the "bochs -f bochsrc.bxrc" command. Kernel.C serves as a test script where both USER SCHEDULER and TERMINATING FUNCTIONS features are enabled to assess their behavior. When the USER SCHEDULER is turned on, it is observed that threads engage in a round-robin sequence in CPU allocation, effectively passing control to one another post-execution of the for loop. With the TERMINATING FUNCTIONS feature active, the test demonstrates that thread0 and thread1 conclude their processes after ten iterations. Meanwhile, thread2 and thread3 are observed to continue execution indefinitely, confirming the persistent operation beyond the termination of the other threads.



Bochs x86-64 emulator, <http://bochs.sourceforge.net/>

USER Copy Paste Screenshot Reset suspend Power

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[92]
 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]
 FUN 3: IN BURST[93]
 FUN 3: TICK [0]
 FUN 3: TICK [1]
 FUN 3: TICK [1]
 FUN 3: TICK [2]
 FUN 3: TICK [3]
 IPS: 27.128M

A: NUM CAPS SCRL

188 // Recursive Fibonacci
 189 // Remem
 190 bool inte
 191
 192 // Disab
 193 if (inter
 194 setIn
 195 }
 196
 197 // Add the thread to the ready queue

Figure 7: Testing