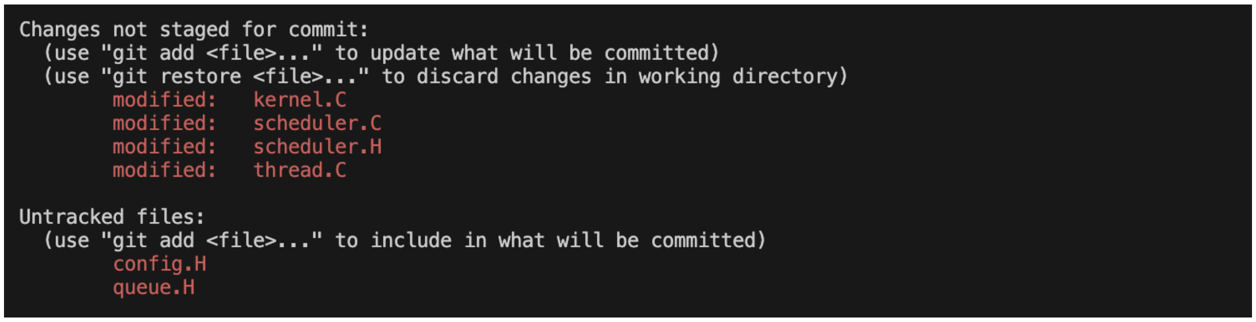
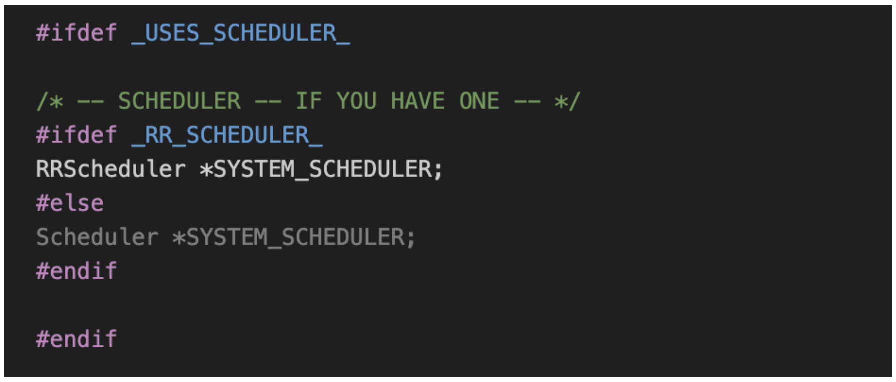
Machine Problem 5: Kernel-Level Thread Scheduling

Changes:

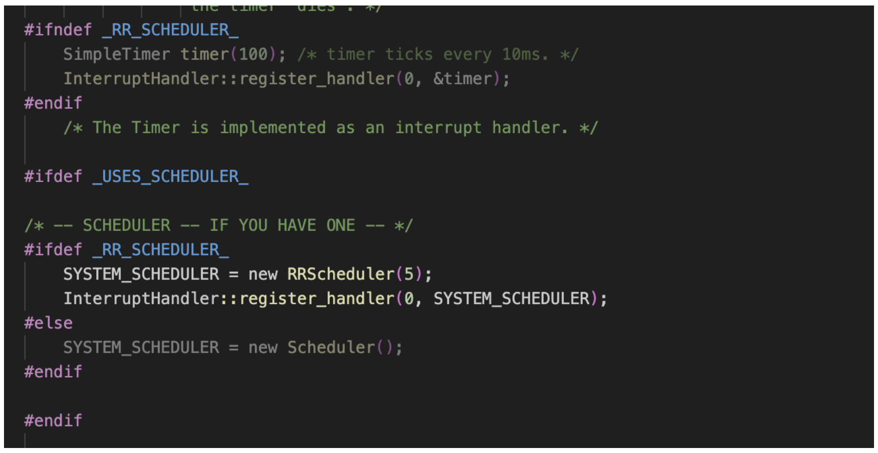


```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   kernel.C
        modified:   scheduler.C
        modified:   scheduler.H
        modified:   thread.C

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        config.H
        queue.H
```

`kernel.C`:
Conditions based on enabling scheduler and type of scheduler



```
#ifdef _USES_SCHEDULER_

/* -- SCHEDULER -- IF YOU HAVE ONE -- */
#ifdef _RR_SCHEDULER_
RRScheduler *SYSTEM_SCHEDULER;
#else
Scheduler *SYSTEM_SCHEDULER;
#endif

#endif
```



```
#ifndef _RR_SCHEDULER_
    SimpleTimer timer(100); /* timer ticks every 10ms. */
    InterruptHandler::register_handler(0, &timer);
#endif
    /* The Timer is implemented as an interrupt handler. */

#ifdef _USES_SCHEDULER_

/* -- SCHEDULER -- IF YOU HAVE ONE -- */
#ifdef _RR_SCHEDULER_
    SYSTEM_SCHEDULER = new RRScheduler(5);
    InterruptHandler::register_handler(0, SYSTEM_SCHEDULER);
#else
    SYSTEM_SCHEDULER = new Scheduler();
#endif

#endif
```

`Scheduler.C` and `Scheduler.H` along with `Queue.H`

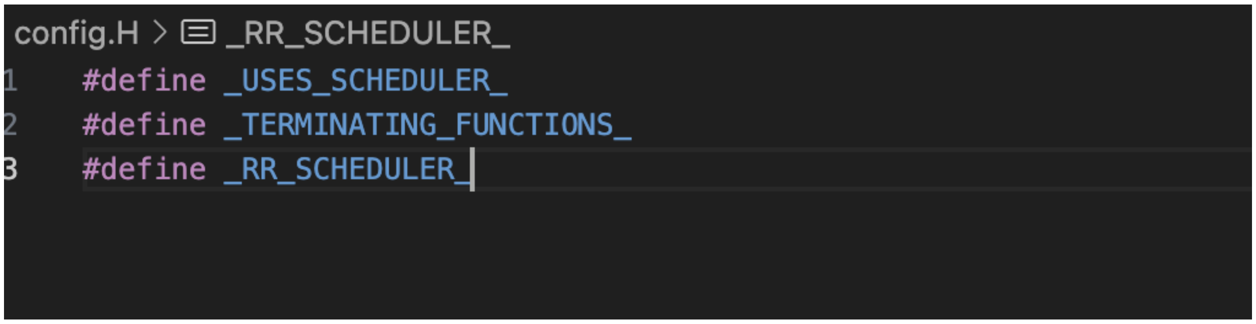
Queue.H defines a class for queue where threads are registered. scheduler uses this queue to perform yield(), terminate() , resume() and add()

`thread.C`

Thread _shutdown when function of thread is completed and thread start by enabling the interrupts

`config.H`

Common file to enable/disable scheduler/type of scheduler and termination of threads



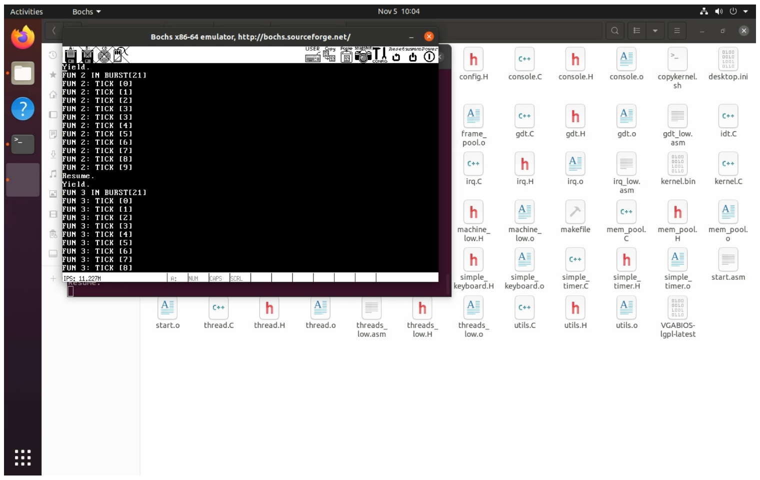
```
config.H > _RR_SCHEDULER_
    #define _USES_SCHEDULER_
    #define _TERMINATING_FUNCTIONS_
    #define _RR_SCHEDULER_
```

Test cases:

FIFO

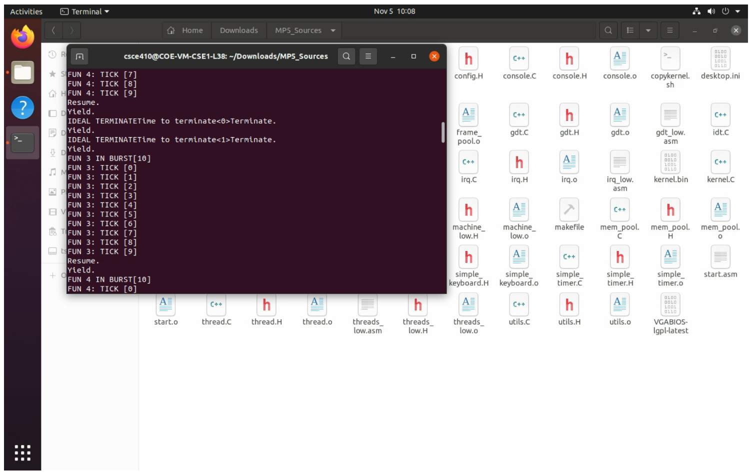
1. In config.H comment `#define _RR_SCHEDULER_`

    No termination



```
Yield.
FUN 2 IN BURST[21]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [3]
FUN 2: TICK [4]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
Resume.
Yield.
FUN 3 IN BURST[21]
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]
```

With termination :



```
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
Resume.
Yield.
IDEAL TERMINATETime to terminate<0>Terminate.
Yield.
IDEAL TERMINATETime to terminate<1>Terminate.
Yield.
FUN 3 IN BURST[10]
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]
Resume.
Yield.
FUN 4 IN BURST[10]
FUN 4: TICK [0]
```

Termination code : Terminate and delete the current thread, give the CPU to next thread in the queue
thread.C

```
static void thread_shutdown()
{
/* This function should be called when the thread returns from the thread function.
It terminates the thread by releasing memory and any other resources held by the thread.
This is a bit complicated because the thread termination interacts with the scheduler.
*/
Console::puts("Time to terminate");
Console::putui(Thread::CurrentThread()->ThreadId());
SYSTEM_SCHEDULER->terminate(Thread::CurrentThread());
delete Thread::CurrentThread();
SYSTEM_SCHEDULER->yield(); // next thread
}
```

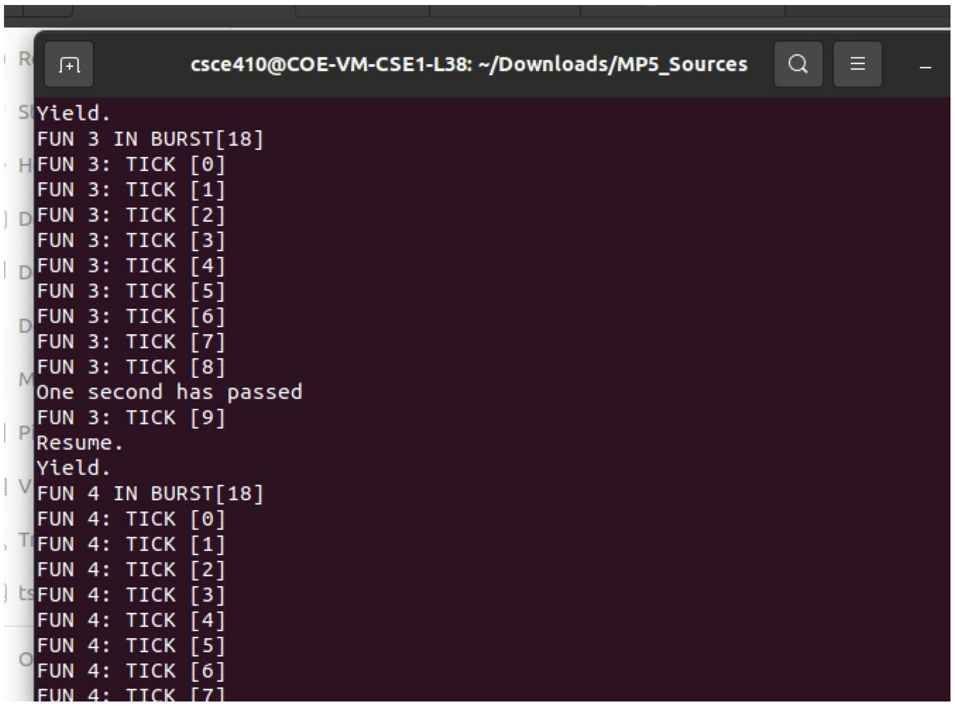
```
void Scheduler::terminate(Thread *_thread)
{

// A->B->C->D-E

// remove C

// start rotating

// B->C->D>-E->A

// C->D->E->A->B

// C Found -> delete

// D->E->A->B deque and enque beyound C

// A->B->D->E

if (Machine::interrupts_enabled())
Machine::disable_interrupts();
// remove the thread from schedular
Console::puts("Terminate.\n");
bool thread_found = false;
if (Machine::interrupts_enabled())
Machine::disable_interrupts();
for (int i = 0; i < queue_size; ++i)
{
Thread *temp = threads_queue.dequeue();
if (temp == _thread)
thread_found = true;
else
threads_queue.enqueue(temp);
}
if (thread_found)
queue_size--;
if (!Machine::interrupts_enabled())
Machine::enable_interrupts();
}
```

Bonuses :

1. Correct handling of interrupts : This is handled by disabling and enabling interrupts before and after any queue operation

```
void Scheduler::yield()
{
if (Machine::interrupts_enabled()) //disable interrupts
Machine::disable_interrupts();
Console::puts("Yield.\n");
if (queue_size != 0)
{
queue_size--;
Thread *next_thread = threads_queue.dequeue();
Thread::dispatch_to(next_thread);
}
if (!Machine::interrupts_enabled()) //enable interrupts
Machine::enable_interrupts();
}
```

Results: Timer interrupt



```
Yield.
FUN 3 IN BURST[18]
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]
One second has passed
FUN 3: TICK [9]
Resume.
Yield.
FUN 4 IN BURST[18]
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]
```

2. Round Robin

A new class inheriting interrupt handler is created in scheduler.C. The functions are similar to FIFO scheduler, with some minor changes

1. Timer : A timer (referred from simple_timer.C) is implemented to keep track of time quantum (50ms)

```
void RRScheduler::handle_interrupt(REGS *_r) // implementation reference
-> simple_timer.C
{
ticks++;
if (ticks >= hz)
{
Console::puts("Time Quantum limit\n");
resume(Thread::CurrentThread());
yield();
}
}

void RRScheduler::set_frequency(int _hz) // implementation reference ->
simple_timer.C
{
/* Set the interrupt frequency for the simple timer.
Preferably set this before installing the timer handler! */

hz = _hz; /* Remember the frequency. */
int divisor = 1193180 / _hz; /* The input clock runs at 1.19MHz */
Machine::outportb(0x43, 0x34); /* Set command byte to be 0x36. */
Machine::outportb(0x40, divisor & 0xFF); /* Set low byte of divisor. */
Machine::outportb(0x40, divisor >> 8); /* Set high byte of divisor. */
}
```

2. After the RR Scheduler object is created, interrupt is registered in kernel.C and disable pass_on_cpu(scheduler will take care)

```
#ifndef _RR_SCHEDULER_
SimpleTimer timer(100); /* timer ticks every 10ms. */
InterruptHandler::register_handler(0, &timer);
#endif
/* The Timer is implemented as an interrupt handler. */

#ifdef _USES_SCHEDULER_
```

```

/* -- SCHEDULER -- IF YOU HAVE ONE -- */
#ifdef _RR_SCHEDULER_
SYSTEM_SCHEDULER = new RRScheduler(5);
InterruptHandler::register_handler(0, SYSTEM_SCHEDULER);
#else
SYSTEM_SCHEDULER = new Scheduler();
#endif

#endif

```

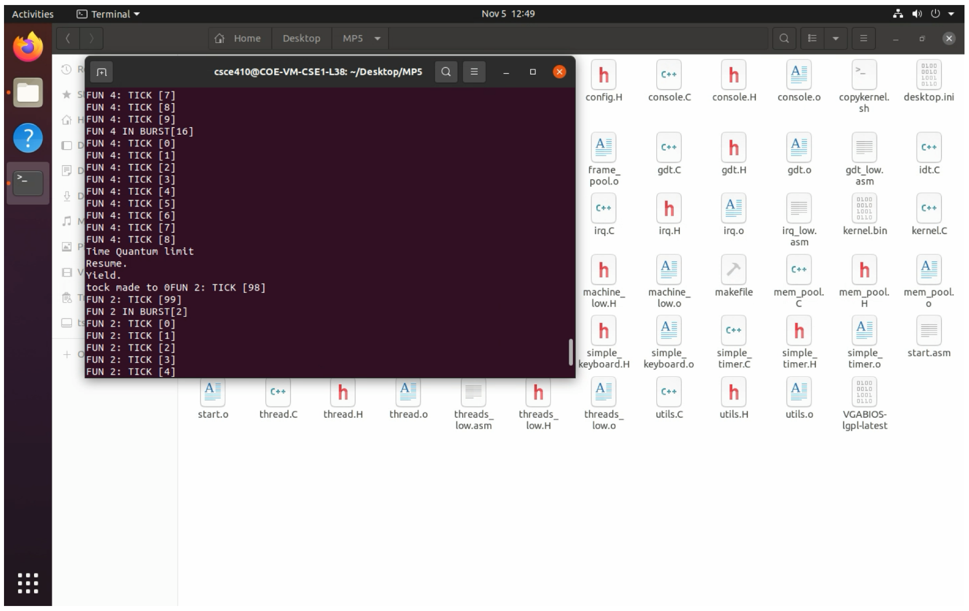
```
{
Console::puts("FUN 4: TICK [");
Console::puti(i);
Console::puts("]\n");
}
#ifndef _RR_SCHEDULER_
pass_on_CPU(thread1);
#endif
}
Console::puts("IDEAL TERMINATE");
}
```

3. yield() -> to reset time quantum when cpu is passed. Also take care of EOI, to indicate the interrupt has been taken care of

```
void RRScheduler::yield()
{
Machine::outportb(0x20, 0x20); // EOI
if (Machine::interrupts_enabled())
Machine::disable_interrupts();
Console::puts("Yield.\n");
if (queue_size != 0)
{
queue_size--;
ticks = 0; /*The 'yield' function must be modified to account for unused quantum
time. If a thread voluntarily yields, the EOQ timer must be reset in order
to not penalize the next thread.*/
Console::puts("tock made to 0");
Thread *next_thread = threads_queue.dequeue();
Thread::dispatch_to(next_thread);
}

if (!Machine::interrupts_enabled())
Machine::enable_interrupts();

}
```

Testing

1. To enable RR scheduler. Uncomment `#define _RR_SCHEDULER_` in config.H



```
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 4 IN BURST[16]
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]
Time Quantum limit
Resume.
Yield.
tock made to 0FUN 2: TICK [98]
FUN 2: TICK [99]
FUN 2 IN BURST[2]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
```