5 DESIGN

CSCE 613

ABSTRACT

Thread Scheduling

Tharun Battula

82400197

This is design doc for thread scheduler for FIFO and Round Robin implementations. How the problem is approached and assumption are made. How interrupts are used in round robin.

DESIGN

In this project you will add scheduling of multiple kernel-level threads to your code base. The emphasis is on scheduling. The below parts have been implemented as per the API specification given Scheduler.H and other project description.

FIFO

I am using an array of Thread pointers to maintain as the ready queue. Using array a ready queue is implemented by rotating array. i.e., initially head is at the 0 and enqueuer is done by adding new thread pointer at the tail and head is moved on. When the tail/head meets the tail the loop is rolled back by doing modulo operations. This way Queue is implemented and it creates the restriction on the maximum number of threads for the kernel but it is impractical to maintain lot of threads beyond certain point.

1) Scheduler()

The constructor is the main part of scheduling data structure. thread_queue = new Thread*[MAX_NUM_THREADS]; currently it is set to MAX_NUM_THREADS is 128. For reading the queue information head pointer and queue length is also maintained. Parameters initializes here to default values 0

2) add()

This function is mainly to add the new threads to the ready queue. The threads are enqueued. In my implementation if number of threads exceeds the maximum number of array supports, it is checked.

3) resume()

This is similar to add process in terms of queuing a process, the current thread goes and gets enqueued in the ready queue as this process is going to get yielded soon.

4) Yield()

This is the current process yielding the CPU to other threads. It mainly takes the head of the current get queue (dequeue) and dispatches the CPU to the selected thread. Simultaneously the head pointer is moved to next in the ready queue.

5) Terminate()

This function is called by the It deletes the thread resources. i.e. it gets stack using the get_stack function implemented and releases memory. Though this is allocated by kernel.c this is freed by this implementation. It deletes the thread as well. Then it calls yield function to select the next head thread waiting in the queue and update queue.

Shutdown of thread

The termiate function is mainly called by shutdown function in thread.c, it is registerd during thread creation as the thread returns its execution this function is called. So our Scheduler::Terminate() is placed inside this code.

Interrupts

- The interrupts are enabled and disabled properly. In the thread_start enable_interrupts() are called since default interrupts are disabled during thread creation.

1. In the interrupt handling code for round robin or voluntarily interrupt pass_on_CPU(). The calls resume() and yield() are called doing scheduling and queue manipulation and dispatching. So to maintain atomicity ie, avoiding interrupts during this switch procedure. This procedure avoids corner cases. Interrupt status checked before calling to avoid assertation errors.
    a. disable_interrupts() called start of resume()
    b. Enable_interrupt() called at dispatching the cpu for next thread.
2. With this interrupt pair in resume() and yield() functions, it creates proper handling of atomicity and avoiding cornor cases. In the case of round robin interrupt handling in the simple timer.c, the notify to IPC(sendEOI function) is coverd under this interrupt pair.

Round Robin

Once FIFO is implemented, by creating an interrupt routine for a periodic quantum of time thread switching can be managed. This round robin is built over the FIFO. In simple Timer, with 100 hz timer, ticks are calculated for every 10ms. A new counter is created and checked until it increases to hz/20 → 50 ms or 5 ticks in the current setup. This interrupt routine is checked every time periodically.

To enable round robin → **#define SCHE_ROUND_ROBIN** in Scheduler.H, by disabling it FIFO is done without round robin

- In the interrupt, the counter is reset first.
- Then using extern variable of SYSTEM_SCHEDULER, resume() is called to enqueuer the current thread for future completion.
- A special care is needed in handling this interrupt since the interrupt changes the thread context. Hardware needs to be aware that this interrupt has been handled so a function send_EOI_to_IPC(_r); is added in the interrupt.C interrupt handler to take the registry values and update EOI message in the hardware low-level. This is a notification to IPC excluding the general interrupt handling type.
- One this happens the thread switch is called → yield() using SYSTEM_SCHEDULER

- It is coded in a way that if the wait queue is empty, the thread need not change. So accordingly the yield function is not called if que length is 0. This case is checked by terminating all the waiting threads and checking the working case.

Code Modifications

File Name	Reason for Modification
Scheduler.C, Scheduler.H	Code for FIFO, add, resume, yield, terminate .H modified declaration of data structure private variables
thread.C and thread.H	Implemented following functions: a. thread_shutdown b. thread_start

simple_timer.C and simple_timer.H	To handle interrupt for Round Robin timer, using the existing code and frequency to utilize it. New variable is created to count the timer for desired 50 ms time period for RR quantum.
interrupt.C and interrupt.H	This was needed to create a segregated function for handling EOI and notifying IPC about the handling of interrupt and skipping the interrupt handling after it resumes the context on the same thread. Interrupt.H has functionality definition. .

The other files modified are (which are not required to submit)

1. kernel.c to debug and implementation purpose
2. console.c → to print the bochs emulator output to a text file stream to view the output and analysis.
3. Makefile.linux64 added scheduler.c and its corresponding make commands for compilation.

Assumptions

- The initializations order is followed according to the kernel.C
    - The definition of threads
    - System Scheduler definition
    - Thread allocation of stack
    - Registering the interrupt service.
    - Adding threads in ready queue

#define _USES_SCHEDULER_

To test scheduling and FIFO evaluation of default implementation.

#define _TERMINATING_FUNCTIONS_

To check the terminating function

- If there is no thread is in waiting queue –
    - If single non terminating thread keeps running → it will stop according without thread switch
    - If there is no thread in the last thread terminates, It reaches assertion failed as the terminating thread returns. Handling of this corner case is out of scope of the project
- Stack memory should be deleted

    - Though kernel.c(programmer) allocates memory for each thread, it Is deleted during the thread (delete). This can be commented if not needed
- There will be racing conditions with the threads to print on the console due to different thread services and mix of printing
- Since the thread routines implement different functions, for simplicity function with non-common resources are considered to avoid critical exceptions on resource incorrect thread behavior on shared variable updates etc. This can be justified by pushing the responsibility to the user to handle such cases using mutex locks or during the coding with proper care.
- For maintaining atomicity at scheduler queue maintenance (resume and yield) and non-interruption during interrupt service, interrupts are enabled and disabled.
- I used a macro #define DEBUG_ENABLE 0 to enable the control over prints.

Sample Round Robin Output:

```
Installed exception handler at ISR 0
Allocating Memory Pool... done
Installed interrupt handler at IRQ 0
Hello World!
CREATING THREAD 1...
esp = 2098608
done
DONE
CREATING THREAD 2...esp = 2099656
done
DONE
CREATING THREAD 3...esp = 2100704
done
DONE
CREATING THREAD 4...esp = 2101752
done
DONE
```

STARTING THREAD 1 ...
Thread: 0
FUN 1 INVOKED!
FUN 1 IN BURST[0]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]Thread: 1
FUN 2 INVOKED!
FUN 2 IN BURST[0]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
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]
Thread: 2
FUN 3 INVOKED!
FUN 3 INThread: 3
FUN 4 INVOKED!
FUN 4 IN BURST[0]
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 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]FUN 2 IN BURST[1]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
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]
BURST[0]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUNFUN 4 IN BURST[1]
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 1: TICK [8]
FUN 1: TICK [9]

FUN 2 IN BURST[2]
FUN 2: TICK [0]
FUN 2: TICK [1] 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]