Red Hat Enterprise Linux Final Report

Ben Andrews, Jimmy Hickey, Erika Jensen

December 1, 2017

Contents

1 Process Synchronization Implementation

1.1 Algorithms & Pseudo Code

We implemented support for both long term and short term critical sections.

If a process only requires a critical section for a short time, our short critical section policy is enacted. On a uniprocessor system, this is handled by turning preemption off. When the process leaves the critical section, preemption is reenabled. On an SMP system, this is achieved with a spin lock.

In the case that a process needs to access a critical section for longer time, binary semaphores are used. When a process requests a semaphore that is already in use, it is put in a queue and is marked as not runnable. When the semaphore is released, the next process in the queue takes it.

```
if process is entering critical section
        if critical section time is short
                disable_preemption
        else
                create_semaphore
```

1.2 Comparison to Red Hat

Red Hat uses both binary and counting semaphores for its long term critical section. Additionally, it has separate support for interrupt critical sections. This is accomplished by splitting interrupt routines in two sections: the top half and the bottom half.

2 CPU Scheduling Implementation

2.1 Algorithms & Pseudo Code

We implemented the Linux `SCHED_OTHER` algorithm. This uses the Completely Fair Scheduler(CFS) and round robin to schedule. Processes can have different arrival times, at which point they will be added to the CFS's red-black tree. The round robin scheduler has a variable time quantum. When initialized, it is given a minimum granularity and a target latency. This is how we determined the time quantum used.

```
time_slice = target_Latency / size_Of_Tree
if time_slice > minimum_Granularity
        give process time_slice
else
        give process minimum_Granularity
```

2.1.1 Completely Fair Scheduler

The CFS uses a red black tree to arrange its processes. A red-black tree is a self balancing binary tree. The CFS uses the *virtual runtime* of a process for insertion. A process's virtual runtime is the sum of its time on the processor. Processes with lower virtual runtime are sorted to the leftward. The leftmost node of the tree is chosen to run on the processor. It is removed and readded when it is kicked off of the processor.

```
Load red-black tree with processes
Choose leftmost child
Pull process off tree
Auto-balance Tree
Run process
Place process back on tree
Auto-balance tree
```

Red-black trees have many desirable qualities. They are self balancing and binary, making them easy to sort, manage and traverse. These traits allow our kernel to quickly and efficiently schedule processes.

2.2 Comparison to Red Hat

Red Hat inherits all of the scheduling algorithms from the Linux kernel. It supports Real-time scheduling with `SCHED_FIFO` and `SCHED_RR` and normal scheduling with `SCHED_BATCH`, `SCHED_IDLE`, and the `SCHED_OTHER` that we implemented. Linux's `SCHED_OTHER` also uses niceness values and priorities to make its scheduling decisions, which we have not fully implemented.

Also, Red Hat specially care for processes that were have exitted an IO burst. These processes are put further left on the process tree than just based on their virtual run time.

3 Memory Management Implementation

3.1 Algorithms & Pseudo Code

We implemented logical paging. We did not map these pages to physical frames, thus we adjusted our page table.

index	page populated
0	True
1	True
2	False
3	False
4	True
5	True
6	False

Since we do not have frames to map to, we are using a Boolean to track if the frame is populated with data.

We are using a 4 kilobyte page size to match the traditional Linux size. Our maximum

memory size will be initialized at startup. This allows us to easily model different systems and generate the appropriate amount of pages.

Upon receiving a process, we have to allocate the proper amount of pages.

```
Receive process
Convert memory from bytes to kilobytes
Determine how many 4 kilobyte pages necessary
Ensure there are enough pages to fit received process
Allocate pages
Unload process from memory when it is done running
```

3.2 Comparison to Red Hat

In contrast, this is how Red Hat links logical pages to physical frames.

index	frame
0	22
1	23
2	26
3	21
4	27
5	24
6	25

frame	data
21	Happy data
22	Grumpy data
23	Sleepy data
24	Dopey data
25	Bashful data
26	Sneezy data
27	Doc data

On top of the page table Red Hat includes a variety of other memory management tools. It also separates memory into four zones and uses slabs to coordinate physical memory.

Red Hat additionally implements virually memory and swapping.

4 Overview of Our Implementation

To start, our code runs on one tick cycles. Generally, one loop of operations on processor takes a single tick. Some exceptions include putting processes into memory and into the

scheduling tree; these were considered timeless activities.

At the beginning of a tick, the kernel checks if any processes have arrived. If a process is found, the memory manager attemps to allocate pages to the process. If there is not enough memory it is put in the "onDeck" queue where it waits until enough memory is available.

Next, all processes that are currently waiting for IO are advanced by one tick. If a process is finished waiting for IO, it is rescheduled with `SCHED_OTHER`.

After the above maintenance is complete, the processor is addressed. If the processor has no process on it, the leftmost runnable process in the CFS's red-black tree is put on the processor. If tree has no runnable processes, the tick continues without running anything on the CPU.

If there is a process on the CPU trying to enter a locked critical section, it is placed into an appropriate waiting queue. It is then marked not runnable.

In the case that the process on the processor is finished running, it is reaped. It is taken off of the processor, its memory is deallocated, and it is not placed back on the scheduling tree. Then, the onDeck queue is checked. If there is now enough memory to for the processes in the front of the onDeck queue, it will be allocated memory and scheduled.

If the process on the CPU is entering an IO burst, it is taken off of the processor and put in the waiting list.

If the process on the processor has been on the CPU longer than the current time slice, it is taken off of the CPU and placed back in the red-black tree. However, if the CPU is not currently preemptible because the current process is in a short critical section, the process runs for another tick. Further, if there is not another runnable process on the tree, the processes is not preempted and is allowed to run.

Finally, if none of the above cases are met, the process continues to run for another tick.

The loop of operations stops if all of the following conditions are met.

- There is no process on the processor.
- There are no processes waiting for IO.
- There are no process in the schedule tree.
- There are no processes in the onDeck queue.

4.1 Output

The console first shows a summary of all the processes that were put into the process list at time 0. The output from the main loop then begins.

A summary block of each tick is printed after the whole tick is processed. The block starts with the tick number. Next, the state of the processes that have arrived at that tick are displayed. Then a sentence describing what happened to the processor is printed. After that, the current state of the processor is printed. Finally, the last line shows the current status of memory.

```
tick: 245
P10 waiting for 1 more ticks
P9 finished processing...process reaped
No process running
Unallocated Memory: 61440/524288KB

tick: 246
P10 waiting for 0 more ticks
No process running...put P13 on the processor
Process Running: P13
Burst Time Remaining: 3
Not Critical Section
Lock: ~
Unallocated Memory: 61440/524288KB

tick: 247
Moving P10 out of the waiting queue
Process Running: P13
Burst Time Remaining: 2
Not Critical Section
Lock: ~
Unallocated Memory: 61440/524288KB
```

Sources

1. IBM developerWorks
2. Linux Kernel
3. Network World
4. Operating Systems Concepts
5. Red Hat Website