OS-CA3

Kasra Noorbakhsh - Kourosh Sajjadi - Mehdi Jamalkhah

810100230 - 810100587 - 810100111

Contents

1 Xv6 scheduling

1.1 sched (Q1)

When sched() is called, it triggers the scheduler to decide which process should run next. The actual implementation of the scheduler is contained in the scheduler() function. Now lets see the program flow :

- **Process Interrupt or System Call**: When a process completes its time quantum, encounters a system call, or experiences some other event that causes it to give up the CPU, the scheduler needs to determine which process to run next.
- **Call to sched()**: The sched() function is called to initiate the scheduling process. This function is often called from various places in the kernel, like the timer interrupt handler or after a system call that yields the CPU.
- **scheduler() Function**: This function selects the next process to run based on the scheduling algorithm (round-robin) and updates the process control block (PCB) or other data structures to reflect the change.
- **Context Switching**: Once the scheduler() function has determined the next process to run, it performs the necessary steps for context switching. This involves saving the state of the current process and restoring the state of the chosen process.
- **Run the Selected Process**: After the context switch, the selected process begins execution.

2 scheduling

2.1 CFS (Q2)

In the Linux Completely Fair Scheduler (CFS), the run queue is represented by a data structure known as the red-black tree. The red-black tree is a self-balancing binary search tree that allows for efficient insertion, deletion, and retrieval of elements in logarithmic time. The CFS uses this data structure to maintain a sorted list of runnable tasks based on their virtual runtime. The red-black tree ensures that tasks with smaller virtual run times (indicating that they are due to run) are positioned closer to the root of the tree, making it efficient to find the task with the smallest virtual runtime.

2.2 Linux VS Xv6 (Q3)

In Linux and XV6, both operating systems have a scheduler for each processing core, and they handle the scheduling of tasks independently on each core.

- **Linux**:

  Sharing Scheduling Queues:

  Benefit: Improved load balancing - When scheduling queues are shared among multiple cores, Linux can easily move tasks between cores to balance the workload. If one core is heavily loaded while another is relatively idle, tasks can be migrated to distribute the load more evenly. Disadvantage: Increased contention - Sharing scheduling queues may lead to contention for the shared data structures, introducing potential bottlenecks and synchronization overhead. Contention can negatively impact performance in highly concurrent scenarios.

- **XV6**:

  Not Sharing Scheduling Queues:

  Benefit: Simplicity and reduced contention - By having separate scheduling queues for each core, XV6 avoids contention issues associated with shared data structures. This can lead to simpler and more predictable behavior, especially in scenarios where tasks rarely migrate between cores. Disadvantage:

Potential load imbalance - Since each core has its own scheduling queue, tasks may not be easily distributed across cores, potentially leading to load imbalance. If one core is overloaded while others are underutilized, the system might not efficiently balance the workload.

In summary, the choice between sharing or not sharing scheduling queues depends on the design goals and requirements of the operating system. Linux's approach of sharing scheduling queues provides dynamic load balancing but introduces contention concerns. XV6's approach of not sharing queues simplifies the design and reduces contention but may lead to load imbalance in certain scenarios.

2.3 For Loop (Q4)

The reason for activating the interrupt mechanism, even in a single-core system, is to trigger a context switch. A context switch involves saving the current state of the CPU for the currently running process and restoring the saved state for the new process. This switch is facilitated by an interrupt.

Here are a few reasons why this interrupt-driven approach is used:

- **Modularity and Portability**: The same scheduler code can be used in both single-core and multi-core systems. In a multi-core system, different cores may be running different processes simultaneously, and interrupt-driven context switching is a common approach.
- **Synchronization**: Even in a single-core system, the interrupt-driven approach helps with synchronization. By using interrupts, the scheduler can ensure that context switches occur at well-defined points, avoiding race conditions and ensuring consistency in the system.
- **Consistency**: If the xv6 operating system is designed to run on both single-core and multi-core architectures, having a consistent approach to scheduling simplifies the code-base. The same scheduler logic can be used in both cases, with the difference being the handling of multiple cores.

2.4 Interrupt (Q5)

In Linux, interrupt handling occurs at two levels: the top half and the bottom half.

- **Top Half**: Also known as the "Fast Interrupt Service Routine (ISR)" or "Task-let." Runs with interrupts disabled. Handles time-critical tasks quickly. Executes as quickly as possible to minimize the time spent with interrupts disabled. Typically, the top half is responsible for acknowledging and handling the interrupt quickly, often deferring non-time-critical work to the bottom half.
- **Bottom Half**: Also known as the "Softirq" or "Task-let." Runs with interrupts enabled. Handles less time-sensitive or more time-consuming tasks. Executed when the system is in a safe state to handle the deferred work. The bottom half is scheduled by the top half to perform actions that can be delayed without affecting the critical interrupt handling process.

Both the top half and bottom half of interrupt handling have higher priority than regular user-space processes. The priority of interrupt handling ensures that the system can quickly respond to external events and manage time-sensitive tasks efficiently. It's important to note that while interrupts have higher priority, excessive or prolonged interrupt handling can impact the responsiveness of the system to user processes. Therefore, it's crucial for the kernel to balance the quick acknowledgment of interrupts with the efficient handling of deferred work in the bottom half to maintain overall system performance.

- **Priority Inversion**: Use priority inheritance or priority ceiling protocols to avoid priority inversion issues. Priority inversion occurs when a lower-priority task holds a resource needed by a higher-priority task, leading to the inversion of their priorities. This can affect interrupt handling as well.
- **Isolation**: Isolate real-time tasks from non-real-time tasks to prevent interference. This can involve running real-time tasks on dedicated cores or using techniques like CPU shielding to protect real-time tasks from interference by non-real-time tasks.

3 System Calls

3.1 Procs-Info

```
$ procs-info
Process_Name    PID     State    Queue   Cycle   Arrival   Priority  R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------------
init            1       sleep    1       52      0         3         1       1       1       0       55
sh              2       sleep    1       1       2         3         1       1       1       0       6
procs-info      3       run      1       0       509       3         1       1       1       0       512
```

Figure 1: printed result

3.2 Foo User-Program

```
$ foo&
$ foo started.
procs-info
Process_Name    PID     State    Queue   Cycle   Arrival   Priority  R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------------
init            1       sleep    1       56      0         3         1       1       1       0       59
sh              2       sleep    1       2       2         3         1       1       1       0       7
foo             5       sleep    1       187     815       3         1       1       1       0       1005
foo             4       sleep    1       0       815       3         1       1       1       0       818
foo             6       sleep    1       187     815       3         1       1       1       0       1005
foo             7       sleep    1       187     815       3         1       1       1       0       1005
foo             8       sleep    1       187     815       3         1       1       1       0       1005
foo             9       sleep    1       187     815       3         1       1       1       0       1005
procs-info      10      run      1       0       2691      3         1       1       1       0       2694
```

Figure 2: foo

3.3 Change-Queue

```
$ change-queue 4 2
1-befor=0
after=2
$ procs-info
Process_Name    PID     State    Queue   Cycle   Arrival   Priority  R_Prty  R_Arvl  R_Exec  R_Size  Rank
-------------------------------------------------------------------------------------------------------
init            1       sleep    1       56      0         3         1       1       1       0       59
sh              2       sleep    0       2       2         3         1       1       1       0       7
foo             5       runble   1       999     815       3         1       1       1       0       1817
foo             4       sleep    2       0       815       3         1       1       1       0       818
foo             6       runble   1       999     815       3         1       1       1       0       1817
foo             7       runble   1       999     815       3         1       1       1       0       1817
foo             8       runble   1       999     815       3         1       1       1       0       1817
foo             9       runble   1       1786    815       3         1       1       1       0       2604
procs-info      12      run      1       0       18685     3         1       1       1       0       18688
```

Figure 3: change queue result

3.4 BFJ Priorities

```
$ change-queue 5 3
0-befor=0
after=3
$ procs-info
Process_Name    PID     State    Queue   Cycle   Arrival   Priority  R_Prty  R_Arvl  R_Exec  R_Size  Rank
-----------------------------------------------------------------------------------------------------
init            1       sleep    1       56      0         3         1       1       1       0       59
sh              2       sleep    0       2       2         3         1       1       1       0       7
foo             5       runble   3       1470    815       3         1       1       1       0       2288
foo             4       sleep    2       0       815       3         1       1       1       0       818
foo             6       runble   0       1483    815       3         1       1       1       0       2301
foo             7       runble   0       1483    815       3         1       1       1       0       2301
foo             8       runble   0       1483    815       3         1       1       1       0       2301
foo             9       runble   1       2283    815       3         1       1       1       0       3101
procs-info      14      run      1       0       42879     3         1       1       1       0       42882
$ change-ratio 5 1 2 3
$ change-ratio-sys 5 1 2 3
$ procs-info
Process_Name    PID     State    Queue   Cycle   Arrival   Priority  R_Prty  R_Arvl  R_Exec  R_Size  Rank
-----------------------------------------------------------------------------------------------------
init            1       sleep    1       56      0         3         5       1       2       3       128
sh              2       sleep    0       3       2         3         5       1       2       3       23
foo             5       runble   3       1470    815       3         5       1       2       3       3771
foo             4       sleep    2       0       815       3         5       1       2       3       831
foo             6       runble   0       1584    815       3         5       1       2       3       3999
foo             7       runble   0       1584    815       3         5       1       2       3       3999
foo             8       runble   0       1584    815       3         5       1       2       3       3999
foo             9       runble   1       2382    815       3         5       1       2       3       5595
procs-info      17      run      1       0       46900     3         1       1       1       0       46903
```

Figure 4: foo

4 Schedulers

We have two versions of this lab one that implemented it using Linked-Lists that is closer to the real scheduler that was requested and another version we have we added a structure in our proc that determines the scheduling queue that our process belongs to then we implemented the different queues.

4.1 Round-Robin

For this we used the same RR that the xv6 OS used

4.2 LCFS

For the version that doesn't uses Linked-Lists we just do a backward foo loop on the process table we could have sorted the arrival times that we implemented them in our proc but based on the RR that was in the xv6 we used that method and just did a backward for loop but in the Linked-List version that is implemented as requested.

4.3 BJF

Just like the others our new structure has the parameters needed for this queue and just like the description we calculate its value and decide on that accordingly.

5 Aging

Just like the description wanted it we added a mechanism that prevents starvation and it does that by checking for a certain amount of time and when it succeeded it we change the queue of our process and we make it better one step.