CS3500: Operating Systems

Course Project Report

An O(1) Scheduler for xv6

Gorre Venkata Satya Praveen
CS18B017

Viswanath Tadi
CS18B047

December 2, 2020

Contents

1 Native xv6 Scheduler

`xv6` has the `Round Robin Scheduler`. The scheduler goes around in an infinite for-loop checking if any one of the processes is in `RUNNABLE` state and simply schedules whenever it finds any. Though functional, the scheduler is naive and does not support priority-based scheduling or behavior-based bonuses for processes.

The scheduler loops through every process in the process list (`xv6` has a fixed number of predefined process structures for simplicity) to find a process to schedule. This can reduce the efficiency if the number of processes in `non-RUNNABLE` states is high. The scheduler can take O(N) time to find a `RUNNABLE` process.

2 The O(1) Scheduler

The O(1) scheduler was introduced in `Linux kernel release 2.6.0`. It was preceded by the O(N) scheduler. As the name suggests, the scheduling algorithm runs in constant time bounds, independent of the total number of processes. Just like its predecessor, the O(1) scheduler is also preemptive and hence supports multitasking. The O(1) scheduler supports priority-based scheduling and implements behavior-based bonuses using certain heuristics. It was superseded by the `CFS` scheduler from `Linux kernel release 2.6.23`, because of its dependence on complex heuristics.

3 Scheduling Algorithm and Heuristics

O(1) scheduler achieves Priority-based scheduling through a pair of identical `multi-level queue` data structures, the active queue, and the passive queue. Each is a list of size `40` whose each element is a linked list, each of which holds the processes pertaining to priorities `100` through `139`.

Whenever a new process is created, it is given a base priority of `120`. Processes can be given different priorities to start with, using the `nice` command. The new process is then added to the corresponding queue (priority 100 in queue 1 . . . priority 139 in queue 40) of the active queue. Whenever the scheduler runs, it finds the first queue (within the active queues) that is non-empty (This operation is essentially O(1) in time because of the finite size of the active queue). It then schedules the first process in the picked queue. Whenever active queue becomes empty, the active queue and the passive queue are swapped.

A process stops running when it's timeslice is over (involuntary timer interrupt) or it performs a system call like exit, wait, sleep, etc (voluntary system call). The process is now placed into the passive queue. It is placed into the linked list corresponding to its dynamic priority calculated as explained below.

The O(1) scheduler has a notion of `dynamic priorities`. Whenever a process behaves like an `I/O bound process` and tends to sleep more, the scheduler gives the process a bonus. The more the process sleeps, the more the `bonus`. On the other hand, `CPU bound processes` are punished by a negative bonus. Higher bonus values allow the process to have lower dynamic priority (closer to `100`) and also higher timeslices, allowing better responsiveness of the system. Bonuses are scaled from `0` to `10`, with `0` being the bonus for the most CPU bound process and `10` being the bonus for the most I/O bound process.

O(1) scheduler has a complex heuristic to calculate the bonus values. The scheduler monitors the `sleep cycles` of the processes closely. It maintains the time for which each process is in `SLEEPING` state and in `RUNNING` state. It calculates a notion of `average sleep time` and scales it to a bonus value.

```
p->sleep_avg += sleep_time;
p->sleep_avg -= run_time;
p->bonus = ((p->sleep_avg * MAX_BONUS) / MAX_SLEEP_AVG);
p->prio = p->stat_prio - bonus + MAX_BONUS/2;  //Value clipped to 100-139
```

The scheduler also supports dynamic timeslices to be allocated to processes. The processes which have a low dynamic priority (close to `100`) get a large dynamic timeslice. The scheduler does this to ensure that the process can complete its `IO burst` and preempt voluntarily. The timeslice allocated to each process depends on only its dynamic priority and is calculated using the below formula.

```
if(p->prio < 120) time_slice = (140 - p->prio) * 20;
else time_slice = (140 - p->prio) * 5;
//time_slice clipped to [MIN_TIMESLICE, INT_MAX)
```

4 O(1) for xv6

The above algorithm for O(1) scheduler was implemented for the xv6 operating system as follows :

- Priorities of a process in xv6 range from `0` to `39`, with priority `0` being the most preferred process for the scheduler and priority `40` being the least preferred.

- New processes are given a base priority of `20`.

- Support for nice command has been added. When running a program through shell, static priority can be initialized to θ as
  `$ my_program -nice` $\boldsymbol{\theta}$

- New system calls `setpriority` and `getpriority` are added to allow programs to change static priority when it runs (useful for test programs) and read dynamic priority as time progresses.

- Bonus is calculated using the following formula. Bonus ranges from `0` to `10`, both inclusive. The formula is simple and can justify itself.

  ```
  bonus = (p->total_sleeptime*11)/(p->total_sleeptime + p->total_runtime);
  p->dynamic_priority = p->priority - bonus + 5; // RHS is clipped to [0,39]
  ```

- Timeslice for a process starts just before the context switch. Hence, time lost in servicing the timer interrupt or scheduling overheads are not included in process's runtime. Timeslice allocated to a process is calculated as follows:

  ```
  if(p->dynamic_priority < 20) timeslice = (40-p->dynamic_priority)*500000;
  else timeslice = (40-p->dynamic_priority)*250000;
  *(uint64*)CLINT_MTIMECMP(cpuid()) = *(uint64*)CLINT_MTIME + timeslice;
  ```

- Necessary locking is used while adding or removing processes from a queue to support multi-core scheduling.

- New processes are added to the active run queue, while preempted processes (due to timer interrupt) are added to the passive run queue. Queues are switched when the active queue becomes empty.

5 Change Log

`kernel/sched.c`

```
-     struct sched_queue{
          struct proc* q[2][40];        // First process* in each queue
          struct proc* qlast[2][40];    // Last process* in each queue
          int sched_active;             // Flag for currently active queue
          struct spinlock lock[2][40]; // Necessary locking structures added
          struct spinlock lock_active; };

-     void sched_init(){}
      // Initialises all queues in both multi-level queues to 0(NULL).
      // Initialises sched_active to 0 which makes q[0] as the active queue.

-     void sched_insert(struct proc* p,int active){}
      // Computes the dynamic priority of the process p using the heuristic.
      // Adds the process p to the corresponding active priority queue
      // if active is 1 otherwise to the corresponding passive queue.

-     struct proc* sched_get(){}
      // Finds the non-empty queue with highest priority in active multi-queue
      // and returns the first process in it. If active queue is empty, make
      // passive queue active and search again.
```

`kernel/syscall.h`

```
-     // Added following entries for system calls
      #define SYS_setpriority  22
      #define SYS_getpriority  23
```

`kernel/syscall.c`

```
-     // Added following entries for system calls
      extern uint64 sys_setpriority(void);
      extern uint64 sys_getpriority(void);
      [SYS_setpriority]   sys_setpriority,
      [SYS_getpriority]   sys_getpriority,
```

`kernel/main.c`

```
-     sched_init();            // Initializing Scheduler structures
                               // Added call to sched_init()
```

`kernel/exec.c`

- Added code to support `-nice` argument through shell `kernel/proc.h` : Added following fields in struct proc

```
-     uint64 priority;             // Static priority of a process
      uint64 dynamic_priority;     // Dynamic priority of a process
      struct proc* next;           // Next proc in the linked list
      int last_queue;              // Last proc in its linked list
      uint64 running_since;        // MTIME stored here before scheduling
      uint64 sleeping_since;       // MTIME stored here before sleeping
      uint64 total_runtime;        // Total time on CPU since forking
      uint64 total_sleeptime;      // Total time slept after forking
```

`kernel/proc.c`

```
-     static struct proc* allocproc(void){}
      // Initialising the process's priority, total_runtime, total_sleeptime.

-     void userinit(void){}
      // Call sched_insert for inserting first process into active run queue.

-     int fork(void){}
      // Call sched_insert for inserting newly created process into active queue.

-     void exit(int status){}
      // Update process's total_runtime.

-     void scheduler(void){}
      // Fetch next process via sched_get(). Configure timer based on dynamic
      // priority of process. Update process's running_since to MTIME.

-     void yield(void){}
      // Update process's total_runtime.
      // Call sched_insert() to add process to passive queue.

-     void sleep(void *chan, struct spinlock *lk){}
      // Update process's total_runtime. Set sleeping_since to MTIME.

-     void wakeup(void *chan){}
      // Update process's total_sleeptime.
      // Call sched_insert() to add process to active queue.

-     static void wakeup1(struct proc* p){}
      // Update process's total_sleeptime.
      // Call sched_insert() to add process to active queue.

-     int kill(int pid){}
      // Update process's total_sleeptime.
      // Call sched_insert() to add process to active queue.
```

kernel/sysproc.c

```
-     // Added definitions for new system calls
      uint64 sys_setpriority(void){}
      uint64 sys_getpriority(void){}
```

kernel/defs.h

```
-     // Added signatures of following functions corresponding to sched.c
      void sched_init(void);
      void sched_insert(struct proc*,int);
      struct proc* sched_get();
```

user/user.h

```
-     // Added following system call signatures
      int setpriority(int);       // Set static priority of self
      int getpriority(void);      // Get current dynamic priority of self
```

user/usys.pl

```
-     // Added following entries for system calls
      entry("setpriority");       // Stubs code for setpriority system call
      entry("getpriority");       // Stubs code for getpriority system call
```

Makefile

```
-     $ K/sched.o \           # Added entry for compiling sched.c
```

6 Link to Video Presentation

`Click here` to view video presentation for the project.

7 References

1. O(1) Scheduler - Wikipedia
2. On the Fairness of Linux O(1) Scheduler - IEEE Conference Publication