Hanh Do Phung
University ID: 14252074
Email: hdd29@drexel.edu

CS 543: ASSIGNMENT 2
Implementing Multilevel Priority Scheduler in Inferno

SOFTWARE ARCHITECTURE:

1. Introduction

The purpose of this assignment is to implement a multi-level priority scheduler in Inferno. The goal for the scheduler is to mimic the algorithm of Unix 6th Edition scheduler, which has 128 levels of priority: the higher the level, the lower priority the process (*) associated with it is to be scheduled. Each time the process finished its time slice, but has not terminated, its priority will be decremented down one level, to the end of the queue in the next level. If it has already been on the last level (priority = 127), it will remain on the same level, but will to append to the tail of the queue. The front of the queue of the highest priority will be scheduled for the next time slice. And to avoid starvation [1], after a period S, all process is raised to the top level and start falling down again. In [1], S is said to be best to be between 10T to 20T, in which T is one quanta. The priority of each function is calculated as in [2]:

$$p = \min\left(127, \frac{c}{16} + 100 + n\right)$$

in which c is a cumulative CPU usage measured from the time the process was last swapped into memory. It is incremented for the currently running process on each clock interrupt up to a maximum value of 255. The value n is a parameter called nice. - [2]

In Inferno, a quanta is not measured in time but in number of instructions. Therefore, the formula above has to be converted into:

$$p = min(127, \frac{c \times u \times f}{i} + 100 + n$$

in which c is now the cummulative number of instructions executed , n is nice, u is the quanta size in Unix (in seconds), f is the frequency of clock interrupts (in Hz) and i is the quanta size of Inferno (in number of instructions). So for this project, an assumption will be made that u=0.1s, f = 60Hz, and i = 2048 instructions. This will leave us with p = min(127, c/340 + 100 + n), (**) which is what will be implemented in the assignment.

The initial starting point for this assignment will be Inferno's original scheduler in a time-sharing round-robin style. Processes are kept in 3 queues: all-process-queue, read-queue, and look-up queue (which is implemented as a hash table based on the process ID). In this assignment, we will only focus on the first two queues.

(*) All processes mentioned in this report will be referring to users space's processes, which in Inferno, will be typed as Prog structures.

2. Approach and Data structures

There are many different ways to implement the goal scheduler. I have thought of using an array of linked list in which the index is the priority of the processes; and processes with the same priority will be in the linked list in chronological order (arrival time will determine the order). However, this comes with unnecessary complexity in implementation of choosing the next process to be given a timeslice as we still have to walk down the array to find the next highest priority process.

Another approach is keeping an unsorted linked list of processes, the same as the one Inferno has already has. The only difference will be the need to look for the process with the highest priority in that linked list to be given a time slice. This comes in the inefficient of having to look through the linked list each time a process is preempted to find the next process to schedule. There is also a complication in choosing which process among the ones with the same priority value as they are not kept in order.

Our approach for this assignment is having a sorted linked list by the processes priorities, from highest priority (lowest priority values) to lowest priority (highest priority values). Each time a process is added to the ready queue, either from coming out of a blocked state or after being preempted and coming back ready, it will be placed exactly between the last process with the same priority and the first process with a lower priority than itself. There is only one exception where a process (p) can never be inserted into the head of the ready queue, only from the second position and above, as we are removing the head of the queue out so p→link cannot point to the (used to be head) anymore. If this is not followed, the program will enter an osblock() function.

3. Software architecture

First, for the scheduler to associate a priority (p in (**)), cumulative number of instruction executed (c in (**)) and nice value (n in (**)) with a process, each process have to store those values in its struct, so members *nice* and *priority* are added in the Prog struct. Since there has already been a member called *ticks* in the Prog value that is not implemented to do anything before, it is now used to store the cumulative number of instruction executed.

In Inferno, there is a global isched structure which keeps track of the queues mentioned in part 1. introduction. The run queue is kept track of using isched.runhd, which point to the head of this list, and isched.runtl to the end. For the implementation using a sorted list, these runhd and runtl pointers will be used to keep track of the sorted list, so no new member for the isched struct is needed.

To get the c value in formula (**), a count variable is added inside the loop inside xec() in xec.c(), in which each instruction in the time slice is loop through. Then the count variable get added into the ticks member of the process in that time slice.

Most of the scheduling activity will be done within the vmachine() function, which will first get the head of the sorted run queue (let say r=isched.runhd) then call xec(r) to give that process a time slice. After the process (r) is done with the time slice (preempted when time is up, or terminated, etc.), vmachine() then recalculate the priority increment its priority value if it is preempted (process falls

down to next priority), move the isched.runhd pointer to the next process in queue (r→link) and insert the process into its right spot in the queue using sortedInsert() function.

Apart from he vmachine, whenever a process is released from a block state to ready state, function addrun will add that process onto the run queue, and also re-calculating priority before inserting using sortedInsert().

To get the nice value, which is set by the user, some changes were made in the devprog.c. In this file, I added a new command on the Cmdtab called setnice, and also enumerated Cmsetnice beforehand. Next , I added a case in progwrite() when write to Ctl (control) file. In this case, p→nice is set to a value parsed by parsecmd() and stored withen a Cmdbuf called cb. The way the user can set the nice value is writing to the correct control file of a particular process. For example, they can do: echo setnice *x* > /prog/n/ctl, and the nice value of x will be set for process n.

The changes were made to these following files:
xec.c, interp.h, dis.c, devprog.c

TESTING and EVALUATION

The scheduler was mostly debugged using a function called printQueuePrior() which prints the ready-queue and the all-process-queue in the format of *pid(priority, nice) → pid(… → …..* This was inserted in vmachine to periodically print the queues out each time modifications is made to the ready queue. The scheduler was tested on functionality by invoking commands with multiple pipes to have many processes on the ready queue. Another approach onto creating lots of processes on the ready-queue was to invoke wm/wm (the GUI of Inferno) and fire up many applications at the same time, especially games that involves a lot of graphics. While these processes run, the printQueuePrior prints queues of processes on the cmd prompt. The result observed is the schedule functions properly, with all the queue kept sorted by priority (except for first process at head of queue), and the programs runs smoothly in Inferno.

Changing the nice value was also tested by typing “echo setnice 20 > /prog/1/ctl” and after the time slice after, 1 (the program that has just been run) was move back to the end of the ready queue (at that time the largest priority value was less than process 1’s current priority + 20). Hence, the nice value is proven to be included in the calculation of priority.

REFERENCE

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA.

[2] Brian Stuart. 2008. *Principles of Operating Systems: Design and Applications*. Course Technology Press, Boston, MA, USA.