Experimenting with MINIX Scheduler

CS4089 Project

End Semester Report

Palavalasa Surya Teja (B120599CS)
Atul Golchha (B120388CS)
Guided By: Dr. K. Murali Krishnan

December 7, 2015

Abstract

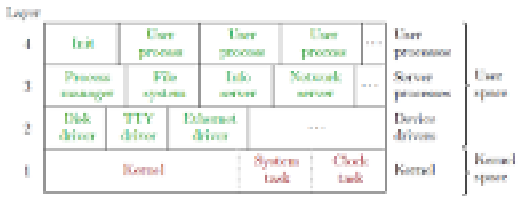
This report presents an introduction to the working of minix scheduler and various components related to it and also a design to replace the present scheduling policy with lottery scheduling.

1 Introduction

Minix is a free, open source, operating system designed to be highly reliable. It has a tiny micro kernel that is running in kernel mode and rest of the process running in isolated, protected user mode. These include the virtual file system, the memory manager, the process manager and device drivers. One consequence of this design is failure of the system due to bugs or attacks are isolated.

1.1 IPC in minix

IPC in minix is done using messages IPC three primitives are provided for sending and receiving messages they are:
1. send(dest, &message);
2. receive(source, &message);
3. sendrec(src_dst, &message);
Each task, driver or server process is allowed to exchange messages only with certain other processes. User processes cannot send messages to each other but can initiate messages to servers and servers can initiate messages to drivers and the kernel.



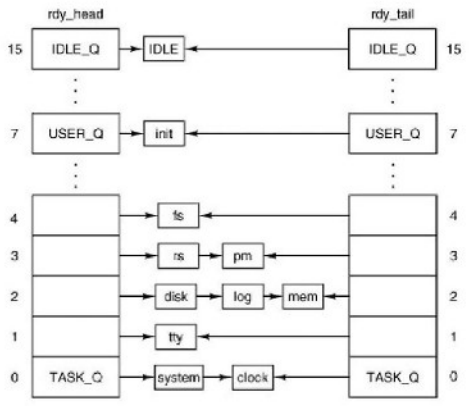
1.2 Process Table

Due to the micro-kernel architecture of minix the process table (as in a monolithic kernel) is split into three different parts each of it is with process-manager, file-manager, kernel. We will be focusing on the kernel part of the process table as this is the place where scheduling takes place. The part of the process table in kernel contains p_max_priority field, which tells which scheduling queue the process should be queued on when it is ready to run for the first time. Because the priority of a process may be reduced if it prevents other processes from running, there is also a p_priority field which is initially set equal to p_max_priority. p_priority is the field that actually determines the queue used each time the process is ready. P_nextready is used to link processes together on the scheduler queues. The two arrays rdy_head and rdy_tail are used to maintain the scheduling queues. The variable p_priv points to the privilege structure which is not part of the process table. It contains fields by which IPC as described earlier is restricted. Only system processes have private copies; user processes all point to the same copy.

1.3 scheduling data-structure

MINIX 3 uses a multilevel scheduling algorithm. Processes are given initial priorities that are related to the structure, but there are more layers and the priority of a process may change during its execution. The clock and system tasks in layer 1 receive the highest priority. The device drivers of layer 2 get lower priority, but they are not all equal. Server processes in layer 3 get lower priorities than drivers, but some less than others. User processes start with less priority than any of the system processes, and initially are all equal, but the nice command can raise or lower the priority of a user process.

The scheduler maintains 16 queues of runnable processes, although not all of them may be used at a particular moment. The below image shows the queues and the processes that are in place at the instant the kernel completes initialization and begins to run. The array rdy_head contains one entry for each queue, with that entry pointing to the process at the head of the queue. Similarly, rdy_tail is an array whose entries point to the last process on each queue. The initial queueing of processes during system startup is determined by the boot_image table in table.c. The queue are numbered from 0 to 15 with 0 being the queue with the highest priority



1.4 scheduling algorithm

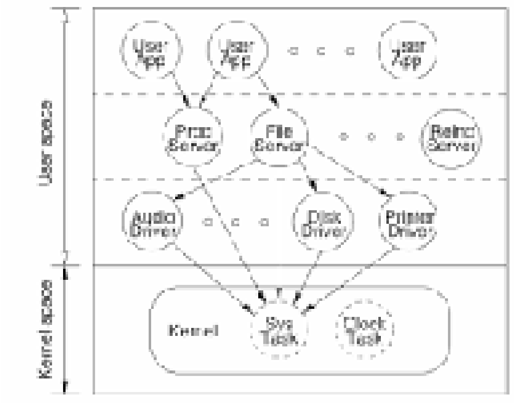
Given the above data structures, the scheduling algorithm is simple: find the highest priority queue that is not empty and pick the process at the head of that queue. The IDLE process is always ready, and is in the lowest priority queue. If all the higher priority queues are empty, IDLE is run. Kernel task priority is not changed.

1.5 lottery scheduling

The basic idea is to give lottery tickets for cpu time at every scheduling a ticket is drawn and the process that is holding the ticket is given the resource To paraphrase George Orwell: "All processes are equal, but some processes are more equal."More important processes can be given extra tickets, to increase their odds of winning. If there are 100 tickets outstanding, and one process holds 20 of them, it will have a 20 percent chance of winning each lottery. In the long run, it will get about 20 percent of the CPU. In contrast to a priority scheduler, where it is very hard to state what having a priority of 40 actually means, here the rule is clear: a process holding a fraction f of the tickets is expected to receive f of the resource in question.

1.6 set_priority

Whenever a set_priority system call is made, it sends a message to the process manager. As we can recall from the earlier discussion, user processes cannot directly communicate with the kernel. The getsetpriority function in the process manager checks weather the user call is valid, if yes it does a kernel call by sending a message to the system process in the kernel to update the priority of the process in the process table. The system process in the kernel has a function corresponding to the message that is do_nice which updates the process priority in the process table of the kernel.



2 Problem Statement

This project aims to get insight about working of minix and familiarizing kernel programming by replacing the present user scheduling policy with lottery scheduling. This involves rewriting kernel scheduling function (as of MINIX v3.1.6 the scheduling is done inside the kernel) and the nice system call that allows a user process to change its priority if invoked by root.

3 Literature Survey

Operating Systems Design and Implementation by Andrew S. tanunbaum and Albert S. Woodhull (third edition) has complete explanation of the working of minix operating system along with source code [1] Carl A. Waldspurger and William E. Weihl paper on lottery scheduling [2] minix3 developer guide [3]

4 Work Done

We studied various components of minix and designed the solution for the above problem statement
The details of the design is given below

4.1 queue

A new queue is added i.e queue no.16 the IDEL_PROC will be running in this queue all the user processes will be queued in the queue no. 15. The following modifications are to be made in the /usr/src/kernel/proc.h file
#define NR_SCHED_QUEUES 17
#define IDLE_Q 16
The first statement increases the number of queues in the scheduling data structure and the second one sets the priority queue of the IDEL process. A new field will be created in the process table structure of the kernel which will contain the number of tickets a process is currently holding in the /usr/src/kernel/proc.h file
char p_priority
char p_max_priority
char p_ticks_left
char p_quantum_size
int p_tickets
The first four lines are some of the fields that are present in the proc structure, we are adding a variable p_tickets that will be used only for user process to store the number of the tickets it owns. we will also be adding a variable
int total_ticks
in the same file in order to track the total number of tickets of all the running processes.

4.2 scheduler

Sched(proc,&q,&front)

The above function in the usr/src/kernel/proc.c file in the kernel has to be modified in-order to implement

the lottery scheduling policy for the user process. The above function when called with a process pointer returns the queue it has to be placed in and weather to add it at the head or the tail.
We will modify the above call so that whenever it is called we will first check weather the process is a system process or a user process if the process is a system process the old scheduling policy will be applied and if the process is user process we will return the user queue number and to add it at its tail.

4.3 pick_process

pick_process()
The above function in the /usr/src/kernel/proc.c file is the function that picks the next process and points the next_ptr to the new process that is to be run. The above function is modified such that the first 15 queues (i.e 0-14) queues that are used by system processes are checked first, if they are empty, then the lottery scheduling kicks in (as mentioned above all the user process are en-queued by sched in the 16th queue i.e no.15). It does not alter the scheduling of the system processes.

If all the 15 queues are empty, the lottery scheduling policy kicks in. A random function is called which picks a number at random between 0 and number of tickets issued till now (as mentioned earlier the total number of ticket is stored in the variable total_ticks in /usr/src/kernel/proc.c). Let the number be x the first process in the queue 16th queue is taken and the x is subtracted from the number of tickets the process posses, if x is negative then the process is scheduled else the loop is run with the next process in queue until x become negative (i.e. a new process is scheduled).

PSEUDO CODE:

```
x = rand(total_tics)
proc = proc_head[16]
while(x >0)
x -= proc.tickets
if(x >0)
proc = proc.next
end while
next_proc = proc
```

4.4 kernel initialization

In /usr/src/kernel/main.c, the code corresponding to boot process is present, where the initialization of tickets for the init process is done.

4.5 enqueue

enqueue(rp)
The above function in /usr/src/kernel/proc/proc.c is called with a process pointer whenever a process goes from blocked state to ready state, the function calls the function sched and adds the process to the corresponding queue as returned by the sched function.
We will modify it to track the total number of tickets possessed by the running process. Whenever a new process is en queued, we update the variable total_ticks.

4.6 dequeue

Dequeue(rp)
The dequeue function in /usr/src/kernel/proc.c does the opposite of enqueue. Whenever a process goes into blocked state, the de queue function removes the above process from the ready queue.
We will modify the above function to decrease the variable total_ticks. Thus, the value gets updated.

4.7 set priority call

set_priority(which, who, prio)
The set_priority system call in /usr/src/lib/posix/priority.c provides a

user library to set priority for a process. The argument which is one of the following PRIO_PROCESS, PRIO_PGRP and PRIO_USER at prest minix implements only PRIO_PROCESS. who indicates the process id and prio indicates the priority. The corresponding system call is converted into a kernel call by process manager, which is handeled by the system process. We will fix a maximum number of tickets that a process can have.

5 Future Work and Conclusions

We would be implementing the above design in the coming semester and also work with on the memory manager if time permits.

References

[1] Operating Systems Design and Implementation (third edition) Andrew S. Tanenbaum, Albert S. Woodhull

[2] Carl A. Waldspurger and William E. Weihl paper on lotter scheduling

[3] minix3 developers guide