User-Level Control of Scheduling
In a Micro Kernel Operating System

by

Aditi Paluskar

November 2001

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science
under the instruction of Dr. Ted Pedersen

Department of Computer Science
University of Minnesota
Duluth, Minnesota 55812
U.S.A.

Abstract

Allocating scarce resources among many users is one of the main tasks of an operating system. Process scheduling refers to the policies which decide the order in which the processes resident on a computer system are executed. The choice of a good scheduling algorithm is a key factor in optimizing the performance of a computer system. However, there is no single best scheduling algorithm as each improves different aspects of overall performance such as throughput, waiting time, response time, etc.

This project focuses on an algorithm which places its emphasis on ensuring fairness among processes. Lottery scheduling allows users to assign relative priorities to processes and then simulates a lottery to select the next process to execute, such that the priority of a process is equated to a number of chances in the lottery.

The objective of this project was to implement a user-controlled lottery scheduling algorithm in the MINIX operating system. This project has enhanced MINIX with a lottery mechanism in the kernel that supports system calls that allow users to control the priority of their processes, and thereby control their scheduling.

Acknowledgments

I am grateful to my advisor Dr. Ted Pedersen for providing me an opportunity to work with him and the valuable guidance that he provided. I also thank Dr. Masha Sosonkina and Dr. Robert McFarland for their co-operation.

I am indebted to my parents, sister and friends, without their support nothing would have been possible.

Contents

1 Introduction

The dictionary meaning of *schedule* is "A production plan allotting work to be done and specifying deadlines" [Com82]. In the real world, we come across various situations where there is a need for scheduling. These situations can be as simple as deciding on the time table for the day or more complex as seen in the scheduling of flights, trains etc.

Scheduling scarce resources for many potential users is a complicated problem in a lot of areas. For example, consider a hospital situation where there is a single doctor and a head nurse. The head nurse acts as a scheduler for the doctor. She is responsible for making sure that the patients get serviced by the doctor in some specific order. The policy which decides this order is very important since there are many patients waiting for treatment and a single doctor. The policy could be as simple as noting the times when the patients arrive, and telling the doctor who has been waiting the longest. However this scheme has a potential drawback in that a patient with a severe problem would have to wait for a long time. The severity of the problem might mean a question of life and death for the patient and hence we need another approach. A second scheme would be where the nurse makes a decision regarding the amount of time it might take to treat a patient. She then allows the patients who need a short time to be treated, to get serviced first. However, this scheme suffers from a long wait for patients who are going to need a longer time for treatment. A third approach would be where the doctor treats every patient for a short amount of time and then moves on to the next one. However this wastes a lot of time for the doctor in moving around. Since there is a single doctor wastage of time is not very desirable. A fourth possibility is where the nurse notes down the severity of the problem for every patient, and then allows the patient with the most severe problem to be serviced first. However in such a scheme if the treatment takes up the entire day, then the doctor will be tied up all the

time and the rest of the patients will not be serviced at all. However, at the end of the day doctor should have attended all the patients who have come in on that day. Now suppose we have two patients with severe problems then, how do we decide which patient gets serviced first? If either of the patients is allowed to go earlier it would be unfair for the other patient and hence we need a scheme which will decide on a good order in such a situation. If the patient is chosen randomly then this could solve the problem since both the patients now have an equal chance of being treated first.

This situation suits many other real world problems where a number of users require service from a resource which is in high demand. In the computer domain, there could be a large number of of users using a computer at the same time. These users require services from the different resources such as the processor, memory and files. The users compete for the use of these resources and hence an order has to be established. The operating system is responsible for managing such resources and in the process allowing the users to execute their application programs smoothly. An application program can be a word processor, an email program or a simple C program. The operating system enforces policies which decides who gets to use a resource, when and in what amount. The resources in an operating system are limited and hence their management is a complex task. This is particularly important since there are a large number of processes which require the service of such resources. A process is an active entity which is eligible to participate in the competition for resources and which can be defined as an "instance of a program in a state of execution" [Mil97]. For example an email program or a text editor are always present on the computer but, only when a user starts them, do they become processes. Among the resources mentioned above, management of a processor is vital.

In a uniprocessor environment, there is a single processor and only one process can execute at a time. However, there may be more than one process waiting for the CPU.

This creates a need for the operating system to enforce rules, known as *scheduling policies* which govern the manner in which the processor is allocated to the various processes. The problems which exist in a hospital are also seen in an operating system. The first solution is one in which the processes are scheduled in the order in which they arrive. This is an example of *first-come first-served* scheduling. The second solution schedules the processes with shorter execution times first. This is known as *shortest job first* scheduling. The third solution in which every process is allowed to execute for a small amount of time, is referred to as *round robin* scheduling in operating system terminology. The fourth scheme which gives higher priorities to more important processes is known as *priority scheduling.* If however there are two processes with the same priority then we need a scheme which will select the processes randomly. This scheme is known as *lottery scheduling* and has the same advantages as in a hospital. The idea was introduced by Carl A. Waldspurger [Wal95].

The goal of this project is to implement lottery scheduling in the MINIX [TW97] operating system, specifically to control the scheduling of processes on the processor.

This is done via a series of system calls that allow a user to control the priorities of processes as they are running.

In the next section we describe the concept of a process and some traditional scheduling algorithms. Section 3 discusses the lottery scheduling algorithm in greater detail while section 4 outlines the structure of the MINIX operating system. In section 5 we describe the approach followed in implementing the lottery scheduling algorithm for the MINIX operating system. Section 6 gives a user manual which describes the usage of the system calls. This is followed by the related work in section 7. Finally, we summarize our conclusions in section 8 also discussing the possible future work. Appendix A gives the trace for the system call and appendix B gives the changes we had to make to implement lottery scheduling algorithm.

2 Process Management

This section describes some basic concepts of a process such as process states, process relationships and process scheduling. All the ideas in this section are well known and commonly discussed in operating systems textbooks. Particularly useful presentations are found in [TW97, Mil97, SJP91].

2.1 Process Concepts

A process is an active entity which competes for the system resources such as the processor, memory, I/O devices and files. Being active, a process is in one of the states such as READY, WAIT or RUN. When the process is started it is in the READY state. In this state, the process owns all other resources except the processor. In the RUNNING state a process has all the resources including the processor and in the WAIT state a process awaits the completion of I/O. A process can undergo various state changes and the operating system keeps track of these changes, and thus the progress of the process. The data structure which contains this information for every process is known as the *process control block*.

When a process owns the processor it is in the RUN state, but it moves either to the READY or the WAIT state when it loses control of the processor. The module of the operating system which decides when a process moves from one state to another is called the *scheduler*. The scheduler decides which process gets control of the processor and the processes change their states accordingly. We are interested particularly in the processor scheduler which is responsible for moving the process from the READY to the RUN state and then out of the RUN state either to the WAIT or the READY state.

2.2 Process Scheduling

In a uniprocessor environment, we have a single processor. However, there may be a large number of processes in various states of execution waiting to be scheduled. These processes can voluntarily give up the processor to other processes so that the others can finish faster, making it a cooperative situation or they can demand more of the processor, making it a competitive situation. Hence a set of policies is required which will decide the order in which the processes should be executed. These are known as *scheduling policies*. The objective of the policy is to make sure that all the processes get a fair chance on the processor and eventually finish in such a way that it does not affect other processes adversely. The degree to which it succeeds in achieving this objective is measured by various criteria which are defined below.

- **Throughput**

  "Throughput is defined as amount of work done in a unit of time" [Mil97]. In our case, we can consider throughput to be the number of processes that complete execution in some measured amount of time.

- **Turnaround time**

  Turnaround time is the difference between the time that a process was started and the time that it finished execution.

- **Waiting time**

  Waiting time is the time that a process spends waiting for the processor.

- **Response time**

  Response time is the difference in time between the start of a process and the time the result appears on the terminal.

- **Fairness**

  Fairness ensures that every process gets to execute for at least some amount of time and that no process has to wait for the processor indefinitely.

- **Processor utilization**

  Processor utilization is defined as the amount of time that the processor remains busy.

2.3 Traditional Scheduling Algorithms

There are a number of traditional scheduling algorithms which try to improve on one or more of the criteria mentioned above. However improving only one factor often results in the degradation of other factors and hence a scheduling algorithm tries to strike a balance between them. We now look at four such algorithms.

1. **First-Come First-Served (FCFS)**

   First-Come First-Served is a simple *non-preemptive algorithm*. A non-preemptive scheduling algorithm is one in which the processor cannot be taken away from a process unless the process gives away control voluntarily. The idea here is to schedule the processes in the order in which they *arrive* in the system. Arrival time is the time when a process becomes ready for execution. The first process to request for the CPU is the first to be serviced. The order of execution is completely decided by the system with no user intervention and hence important jobs are given no special treatment by scheduling first. In terms of performance criteria however, this algorithm reduces the number of *context switches*, a condition in which the processor is taken away from a process and handed over to the next. However it results in long waiting and turnaround times for shorter processes, if a long process arrives first. This in turn increases the average waiting and turnaround time.

2. **Shortest Job First (SJF)**

   The disadvantage faced by the short processes in FCFS can be solved by letting the shorter processes execute first. This is the idea of the shortest job first algorithm. The execution times of the processes decides the order of scheduling and is again controlled entirely by the system. Prior knowledge of the execution time of processes is required and this can involve additional computation which is an overhead on the system. However, this algorithm reduces the average waiting time of the processes.

3. **Round Robin (RR)**

   Round Robin is a strictly *preemptive algorithm.* A preemptive algorithm is one in which the scheduler can take away the processor from one process and assign it to another process. The algorithm improves the fairness criteria. This is achieved by allowing every process to execute only for a fixed time interval known as the *time slice.* At the end of the time slice a new ready process is selected for execution. This improves the response time of short processes since they usually finish their execution in one time slice but decreases the response time of longer processes, since they usually need more than one time slice to finish execution. Observe that the processor is busy all the time. This improves utilization but at the cost of an increase in the number of context switches. The scheduling decisions are made by the system, with hardware support in terms of an interval timer. This timer sends an interrupt at the end of every time slice. MINIX uses RR for scheduling user processes.

4. **Priority Scheduling**

   None of the algorithms discussed above take into consideration the importance of the different processes. When a highly important process is ready for execution, it should be serviced first. Priority scheduling algorithm solves this problem.

The algorithm gives the user control in making the scheduling decision based on the importance of the different processes. This is necessary because the completion of some process earlier may speed up the execution of other processes and thus allow for the faster completion of all the processes. This is achieved by assigning priorities to processes. The process with the highest priority is always chosen to run. The priority of a process can be decided by the user or by the system depending on the various criteria such as the time needed for execution etc. This algorithm can be both preemptive as well as non-preemptive. However, this algorithm suffers from a condition known as *starvation*, which occurs when low priority processes wait indefinitely for the processor. However, a scheme known as *aging*, which increases the priority of low priority processes over time. This change in priority is however brought about by the system. Lets consider a system in which aging is not implemented and there are 4 processes with priorities 10, 2, 1 and 5. With a priority scheduling scheme the process with priority 10 is always scheduled first and the remaining processes will have to wait until it finishes execution. If the process has a long execution time then the remaining processes will have to wait indefinitely and this will cause starvation. Now suppose, the priorities of the processes were 10, 10, 10 and 11. In this situation again the process with priority 11 will always get scheduled first even though the remaining three processes have a priority almost equal to the process being scheduled. We are left with two questions: *Is this fair? Is there a way to get around this problem?*

In the next section, we will look at a scheduling approach which answers these questions.

3 Lottery Scheduling

In this section we look at a different scheduling approach known as *lottery scheduling.*

3.1 Why Lottery Scheduling?

Consider 4 processes with priorities 10, 10, 10 and 11. Priority scheduling will always schedule the process with priority 11 first. The problem with priority scheduling is that it always selects the highest priority process and the low priority processes might have to wait for an indefinite amount of time. This problem of starvation can be solved by using the lottery scheduling scheme in which the low priority processes also get a chance to execute. This is the idea behind lottery scheduling. A discussion on the details of this policy follows. A more extensive discussion can be found in [Wal95].

3.2 Lottery Scheduling Algorithm

Lottery scheduling is a variant of priority scheduling. Like priority scheduling, lottery scheduling also assigns priorities to the processes. The priorities are modeled as *tickets.* Every process owns a certain number of tickets. The higher the number of tickets that a process owns, the higher its priority. The selection of a process for scheduling is done by choosing a random ticket number. The process with this ticket number is scheduled to execute. Thus the term *lottery.* Only the processes which are ready to execute are allowed to participate in the lottery. A random ticket number is chosen from the set of tickets assigned to the ready processes. We will henceforth refer to this term as the *active set* of tickets. Randomness of the process selection is the key idea of lottery scheduling and this idea differentiates priority scheduling from lottery scheduling. This idea ensures that the low priority processes with less number of tickets *have some chance* of being scheduled and

the high priority processes *always have a greater chance* of being scheduled. Consider the example of 4 ready processes with 10, 10, 10 and 11 tickets in this scenario. There are 41 tickets in the active set of tickets. A ticket number between 1 and 41 is therefore chosen and the process with that ticket number is scheduled. Since all the processes have an almost equal number of tickets, they have almost equal chances of being scheduled. We can replace the concept of a "chance" with that of a probability. In terms of probability, the 4 processes will have probabilities of 10/41, 10/41, 10/41 and 11/41 of being scheduled. This means that all processes have an almost equal probability. This overcomes the disadvantage of priority scheduling where the process with priority 11 was always the one to be scheduled and ensures fairness among almost equal priority processes.

Now consider the example where there are 4 ready processes with 1, 2, 3, and 10 tickets. Here the probabilities of being scheduled for the processes are 1/16, 2/16, 3/16 and 10/16 respectively. We can see that process with a single ticket has the lowest probability but at least it has a 1 in 16 chance of being scheduled. If aging was not implemented in priority scheduling then the processes with priorities 1, 2 and 3 would have to wait until the completion of process with priority 10. Thus, lottery scheduling solves the problem of starvation in an environment where aging in not implemented. However, when aging is implemented the system can increase the priorities of the low priority processes. Lottery scheduling gives control to the user in changing the priority of a process. This is done by changing the number of tickets allocated to that process. This feature is provided to ensure that processes, which are more important for the user, are allowed to run earlier. A process maybe more important for a user for various reasons such as its earlier completion will ensure the faster completion of the other processes or the results obtained from that process might be required for the execution of some other process. For example consider two processes with an initial ticket allocation of 1 each. They have an equal probability of

1/2 of being scheduled. If now the tickets of the second process were changed to 10, the probabilities of the two processes being scheduled would change to 1/11 and 10/11. This would give the second process a higher chance of being scheduled. In addition to increasing the probability of being selected in one lottery, an increase in the number of tickets also increases the number of times that a process will get scheduled. This means that a process with a higher number of tickets gets executed more often and hence gets more time to execute on the processor. For example, if there were two processes with tickets 1 and 10 respectively, then after a fixed time period the process with 10 tickets will be scheduled 10 times more often than the process with a single ticket. The number of times that the process gets to execute is also decided by the load on the system. If there are a large number of processes, then every process will have a lesser chance to execute. To illustrate this fact, consider 2 processes, 1 of which has a single ticket and one the other has 10 tickets. The process with 10 tickets has a probability of 10/11 to get scheduled. Now suppose 4 more processes enter the system each with a single ticket. The active set of tickets is now 15 and hence the process with 10 tickets has a probability of 10/15 to get scheduled. This is lower than the initial probability of 10/11. Thus, the worth of a ticket is inversely proportional to the load on the system. If the system is heavily loaded then the worth of a ticket decreases and vice versa.

These features of lottery scheduling make it a very responsive scheme. The Round Robin scheduling for user processes in MINIX does not have a provision for prioritizing the processes. We therefore implement lottery scheduling. The next section gives the details of how the scheduling scheme for MINIX works.

4 MINIX

We have implemented the lottery scheduling algorithm for the MINIX operating system and hence in this section we look at some background material on MINIX. We will give a brief description of the internal structure of MINIX followed by the scheduling policy of MINIX.

4.1 Introduction

The MINIX operating system is written by Andrew S. Tannenbaum [TW97]. It is a small operating system, yet has all the functionality of any other bigger operating system. MINIX is an implementation of the UNIX. It is a micro kernel operating system and has a layered organization. The different modules of the operating system such as the memory manager, file manager, clock etc. are arranged in different layers according to their order of importance and their functions. This layered organization hides the lower level implementation details from the higher layers and makes it easier to make changes to or add new functionality to the system. A detailed documentation on the MINIX operating system can be found in the operating system book written by Tannenbaum [TW97]. It also contains a complete listing of the kernel code with explanations of the various important sections of the code. In the following section we discuss briefly the internal structure and the scheduling approach of MINIX.

4.2 Internal Structure of MINIX

MINIX has a layered organization. It is divided into 4 layers and each layer has a specific task. Also there is a clear abstraction of any layer from the internal details of any of the lower layers. This gives it a much simpler interface. The question which arises from

such a structure is: *How does a module in one layer interact with another module?*. The answer is a communication technique called as *message passing*. Message passing is a technique which facilitates communication between processes ensuring synchronization between the processes. In the MINIX operating system the modules from the different layers communicate by passing a message which contains the data to be exchanged along with the destination and the source of the message.

The separation of the modules into the different layers is done according to the similarities and the differences in the functionality of the different modules. So the modules which perform similar tasks are grouped together. Also as we reach higher layers the importance of the modules decreases with the highest layer having the processes with the least priority.

We will look at each of these layers now. For a more detailed description of each layer please refer to [TW97].

The layer which is lowest in the hierarchy is responsible for handling the low level details of the system such as servicing of interrupts and process management. It also handles the message passing interface which is the medium of communication in MINIX. This layer abstracts the low level details of the system from the higher level processes. The second layer in the hierarchy handles I/O. The processes which perform this function are known as the *tasks*. Examples of tasks are clock task, disk task etc. These processes are serviced after the layer 1 processes. There is one such task known as the *system task* which does not perform I/O but it acts as an interface to provide low level services to the processes at a higher level. The third layer in the hierarchy contains the processes which are known as the *servers*. Examples of servers are memory server, file server etc. As the name suggests they serve the user processes to provide basic functions for their smoother execution. The highest layer consists of the user processes such as executable programs, email program, word processor etc. The priority of the processes decreases as we move higher in the

hierarchy with the tasks having the highest priority and the user level processes having the lowest priority.

The processes in every layer communicate with other processes from the same or different layer through message passing. The process from higher layers depend on the services provided by the lower layers for their execution. Apart from these four layers the kernel holds a special position in the organization. This is because the kernel is a program that does not belong to any layer in particular but it is a result of linking the tasks and the layer 1 code. The kernel contains the routines which perform process scheduling. We change a part of this scheduling code to accommodate lottery scheduling. The servers interact with the kernel via the system task by message passing.

4.3 Process Scheduling in MINIX

The layered organization of the MINIX operating system is the key in determining the scheduling scheme for the system. A traditional scheduling algorithm which combines one or more traditional scheduling algorithms is *multilevel queuing*. This algorithm can be used when the processes can be naturally grouped into different categories depending on their functionality. The processes are divided into different queues. Each queue is given a priority level and priority scheduling is used to schedule the queues. Also within each queue a different scheduling scheme can be used. A process which is in a higher priority queue is allowed to run before any of the processes in the lower priority queues.

Since the different processes in MINIX are organized into layers according to their functionality the multilevel queuing scheme applies very easily. There are 3 different ready queues, one each for the task, server and the user processes. The TASK queue is given the highest priority followed by the SERVER queue and then the USER queue. This is in accordance with their importance level.

The scheduling scheme adopted in the TASK queue and the SERVER queue is FCFS [TW97] whereas the user processes are scheduled using round robin [TW97]. Since the highest priority queue is the TASK queue, a server or user process will be allowed to run only if this queue is empty. Similarly a user process will be allowed to run only if both the SERVER queue and the TASK queue are empty.

The system also maintains two additional data structures for managing the queues. There are two arrays `rdy_head` and `rdy_tail` which keep track of the processes at the head and the tail of each queue. Each of these arrays has 3 entries one for each of the 3 queues. The entries in the first array contain pointers to the processes at the head of each queue whereas the entries in the second array contain pointers to the processes at the tail of each queue.

At the end of each time slice a new process has to be chosen to execute and has hence the scheduler is invoked by the CLOCK task. The scheduler first checks to see if there are any processes on the SERVER queue and if so selects the process at the head of the queue. The entries in the `rdy_head` and `rdy_tail` are updated accordingly. If there is no process on this queue then the scheduler checks the TASK queue. It repeats the same procedure for this queue. If both the queues are empty then the scheduler checks the USER queue. If the process at the head of the queue has completed its execution then the entry for the process is removed from the queue and the next process is chosen for execution. The entries in the arrays are modified accordingly. If the process has not finished its execution then the process is put at the end of the queue and the next ready process in the queue is chosen for execution. A new ready process which becomes ready for execution is always put at the end of the queue. If all the 3 queues are empty then a special process called as the IDLE process is allowed to run.

Thus the tasks and servers are allowed to run to completion while the user processes

are allowed to run until the time slice expires. At the end of the slice a new ready process is chosen for execution.

However, RR algorithm is not an optimal algorithm for user processes as it does not assign priorities to these processes. The user has no control in making the scheduling decisions an hence lottery scheduler makes a better choice for scheduling of processes in this layer.

5 Approach

This section describes the approach that we have adopted in implementing the lottery scheduling algorithm. The section describes the implementation details and the changes made to the various parts of the kernel code. We provide a user manual which gives the details and usage of two system calls.

5.1 Implementation Approach

Our focus is on the user level processes and therefore we replace the round robin scheduling approach to that of lottery scheduling. To implement lottery scheduling we make changes in various sections of the code. We will consider these changes in terms of three broad categories such as tickets, lottery scheduling algorithm and the assignment of tickets.

- **Tickets**

  Ownership of tickets is the most important feature of lottery scheduling. Each process owns a certain number of tickets which decides its priority level. The processes need to keep track of their current ticket allocation at all times. We incorporated this feature into the existing system by including an additional field in the process control block of every process. Thus in addition to the rest of the scheduling information every process now maintains information about the number of tickets its owns. By including this field in the process control block it is possible to keep track of the changes in the number of tickets as well.

- **Lottery Scheduling Algorithm**

  The MINIX operating system makes a scheduling decision at the end of every time slice and selects a new ready process to execute. We do not modify this feature of MINIX.

In our implementation, as well, the scheduler is invoked at the end of every time slice by the CLOCK task. The multilevel queuing implementation of MINIX is also retained. Lottery scheduling is implemented only for the user level processes and hence the tasks and servers are allowed to run at their own priority levels to completion. The scheduling function will make use of the lottery scheduling algorithm only when there are no tasks and servers which have not been serviced yet. The scheduler will find out the number of tickets for all the processes on the USER queue and from that determine the active set of tickets. A random ticket number is then selected by using a random number generator function like `rand()`. The USER queue is then traversed to locate the process which has the winning ticket number. This is simply done by adding up the tickets for every process successively until the sum is greater than the winning ticket number. The process with the winning ticket number is the next to be scheduled and hence it is brought at the head of the queue. The corresponding entry in `rdy_head` is updated so that it now points to the newly selected process. The entry in `rdy_tail` is also updated so that it now points to the process which is at the end of the queue. MINIX uses a process pointer which always points to the currently running process. This pointer is also updated to point to the newly selected process.

- **Assignment of tickets**

  The number of tickets assigned to a process decides the priority of that process. Increasing/decreasing the number of tickets results in increasing/decreasing the priority of the process. Tickets can be assigned by the kernel or by the user. We implemented both the possibilities. A process is assigned tickets by the kernel when it is created. For simplicity we assigned just a single ticket to every process. This was done by modifying the FORK system call. To implement the assignment of tickets under user

control we implemented a system call, `settickets()`. We also included an additional system call, `gettickets()` which will determine the current ticket allocation for a process. The `settickets()` system call takes as a parameter the process identifier and the number of tickets. It then modifies the ticket allocation for that process with the new ticket allocation value. The call originates in the user level process and then flows into the different layers of the system. More specifically the layers which are modified are the memory manager and the system task. The user level process passes the information about the tickets and the process identifier to the memory manager in a message which the memory manager just forwards to the system task. The system task is responsible for the interaction between the memory manager and the kernel. The system task looks through the process table to find the process with the process identifier value specified in the message. It then modifies the ticket allocation value for the that process. The result of this change is then propagated back to the user process through reply messages. The call can also be traced through the other layers because of the various interrupt handling and signal handling which takes place. A complete trace of the system call can be found in appendix A. The changes made to the various section of the code to accommodate the lottery scheduling scheme are outlined in appendix B.

5.2 User Manual

This section gives the description and the usage of the system calls with examples.

- `tic_t gettickets(pid_t pid)`

  Description

  This system call is used to get the ticket count for a particular process. The system call takes as an argument the process identifier (pid) and returns the ticket count. The ticket count type (tic_t) is defined to be of type `int` in the */include/sys/types.h*. The */src/mm/getset.c* file contains the definition of the function. The function looks for an entry in the process table whose pid is the same as that passed by the user as an argument and then returns the ticket count for that process.

  Returns

  The function returns `tic_t` which is the ticket count. `tic_t` is defined to be of type int.

  Usage

  To get the ticket count for a child process with process id *childpid* we can use the gettickets() in the following manner:

  childtickets = gettickets(childpid);

  The number of tickets for the child process will be returned in *childtickets*.

  Example

  We need to find the number of tickets assigned to a process with pid = 10, then we can do so by the following system call:

tickets = gettickets(10);

`tickets` will contain the number of tickets for the process with pid = 10.

- `int settickets(pid_t pid, tic_t tickets)`

  Description

  This system call is used to set the ticket count for a particular process. The system call takes as arguments the process identifier (pid) and the number of tickets (tickets) which need to be assigned to a particular process and returns the status of the assignment (1 if OK). The *src/kernel/system.c* contains the definition of the function. This function looks through the process table entries for the process whose pid is specified in the message. It will then set the ticket count for that process with the new value specified by the user.

  Returns

  The function returns `int` which is the status of the call. If the call is successful then it returns 1.

  Usage

  To set the ticket count for a child process with process id *childpid* to 100 we can use the settickets() in the following manner:

  status = settickets(childpid,100);

  The `status` will be 1 if the call is successful.

  Example

  To set the ticket count for the process with pid = 10 to 200 we can use the following system call :

status = settickets(10,200);

The call will return `status` 1 if the call is successful.

The following code illustrates another example where a process forks off a child process. The parent process is then assigned 100 tickets and the child process is assigned 10 tickets. This is done by the `settickets()` system call. The `gettickets()` system call is used to show that this assignment is correctly done.

```
#include <minix/config.h>/* MUST be first */
#include <ansi.h>/* MUST be second */
#include <sys/types.h>
#include <minix/type.h>         //For message
#include <minix/syslib.h>       //For send
#include <stdio.h>

int main()
{
  int fork_pid;
  int child_tickets, parent_tickets;
  int child_pid, parent_pid;
  int value;
  fork_pid = fork();

  if(fork_pid == 0)
  {
      printf(" I am the child");
```

```
        child_pid = getpid();
        value = 10;
        settickets(child_pid,value);
        child_tickets = gettickets(child_pid);
        printf(" Process Id is : %d, Value of tickets is %d",child_pid, child_tickets);
    }
    else
      {
        printf(" I am the parent");
        parent_pid = getpid();
        value = 100;
        settickets(parent_pid,value);
        parent_tickets = gettickets(parent_pid);
        printf(" Process Id is : %d, Value of tickets is %d",parent_pid, parent_tickets);
      }
}
```

6 Related Work

In this chapter we look at similar work which has been done in the area of lottery scheduling.

Carl A. Waldspurger [Wal95] introduced the idea of lottery scheduling. One of the aims was to design an algorithm which would take into account the characteristics of the application and assign priorities to them accordingly. A second aim was to allow the user to assign and modify the priorities of the processes dynamically. The scheme also allowed the processes to execute for a time which was proportional to their priority. Lottery scheduling idea is a variant of priority scheduling. The original algorithm allowed for modification of ticket allocation. This was done in two ways, either by an explicit *ticket transfer* between two processes or by *ticket inflation/deflation* [Wal95]. The idea behind ticket transfer was to speed up the execution of a process if there was another process awaiting results from the first one. In this situation the waiting process could transfer its tickets to the second process so that it finished execution faster and send back the results. The waiting process would then reclaim the tickets and continue its execution with the tickets and the results obtained from the second process. Waldspurger gives the example of a remote procedure call(RPC) to explain this idea. Ticket inflation/deflation is a scheme by which the ticket allocation of a process could be increased/decreased to ensure that all the processes execute to completion faster and in turn ensure the faster completion of the job. The ticket modifications gave the user the capability to modify the priorities of the different applications.

The algorithm introduced by Waldspurger provided the basic idea which was tested in the Mach 3.0 micro kernel operating system [WW94]. However it was not tested across other operating systems which operate under different workload conditions. This algorithm was then extended so that it performed well on the FreeBSD operating system [PMG] and Linux [SS99]. Additional features such as exchange of resource specific tickets [SHS99]

were provided to further improve the performance and provide additional user control.

7 Conclusions and Future Work

This section makes some observations about lottery scheduling and MINIX and draws some conclusions based on our implementation of the former. It also outlines possible future work in this area.

7.1 Observations and Conclusions

We implemented the lottery scheduling algorithm as it prioritizes processes and gives the user control in assigning the priorities. At the same time it ensures fairness.

The lottery scheduling scheme has been tested on various operating systems such as the Mach 3.0 [Wal95], FreeBSD [PMG] and Linux [SS99]. We implemented this scheme for the MINIX operating system. MINIX is a micro kernel operating system with a simple kernel since the other functionality such as managing of memory, files etc. is separated from the kernel. The source code is well organized and well documented. During our study of MINIX, we got a chance to become a part of the larger "MINIX community" which maintain their own mailing list. We got a chance to learn and participate in discussions about the various aspects of MINIX. When studying MINIX we learned about SMX [Ash96], a Solaris version of the MINIX operating system. SMX runs as a user process on Solaris and thus allows multiple copies of the operating system to run on the same machine. Since SMX runs as a user process, it is scheduled for execution just like any other Solaris process. Other resources such as memory and files are also managed in a manner which is similar to other processes. We decided to choose SMX as the platform for implementation since the re-compilation of code in SMX meant simply invoking the MINIX process on the Solaris machine.

Waldspurger [Wal95] mentions in his work that the idea of lottery scheduling can

be extended over other resources such as memory, locks, I/O devices etc. In memory management the simple paging algorithms could be replaced by such a scheme. We aimed at exploring this possibility in the MINIX operating system. However a further study of MINIX showed that the system makes uses of segmentation and not paging to manage memory. Hence it was not feasible to use this scheme for memory management. We then tried to locate other possible queues in the system where processes wait for different resources. If any such queues exist then, they would be using some scheme to schedule the processes and we could try changing this existing order to that of lottery scheduling. However, we were unable to identify such queues and hence we decided to shift our focus to process scheduling and in particular the scheduling of user processes.

A study of the MINIX code, showed that the process scheduling code is written in the kernel section of the system. We changed this section of the code to accommodate the lottery scheduling algorithm. To implement the key feature of assigning priorities, we added two system calls. The addition of system calls relies completely on the message passing interface which is the communication technique used in MINIX. The user processes communicate with the memory manager and this module in turn can communicate with the kernel through the system task. We made use of such an exchange in which the the user passes the information about the tickets from the Memory manager which in turn conveys it to the kernel via the system task. These system calls have added more functionality to the existing system by giving the user the ability to control the priorities of the processes. To test the system we needed to have real world test data. However SMX being a very small operating system and not a very complete one, this was not possible because of its inability to run large application programs simultaneously. We aimed at measuring the performance of the algorithm by comparing the various criteria like wait time, throughput, turnaround time etc. of lottery scheduling and the existing round robin scheduling algorithm. However

SMX lacks a performance monitor tool like “top” in Unix which would have been able to provide us with sufficient information for calculating the performance criteria. We then decided to make use of just the ”ps” command which gives a snapshot of the system state instead of a dynamic view. However, there were some problems we encountered while executing this command in SMX, which we have now fixed.

7.2 Future Work

We have provided the basic functionality for lottery scheduling in SMX. However, there are a lot of features which could still be implemented to enhance the performance of the system. We have provided for ticket assignments, but it is possible to implement ticket exchange among processes. This would mean additional system calls which will allow the user to increase/decrease the number of tickets for a process by a fixed value. Message passing can be used to implement the idea of ticket transfers between processes. However in implementing these schemes care should be taken that a process does not increase its tickets to a very large amount and starve other processes. The management of tickets has to be done in a very careful fashion and one possible way to achieve this is to think of tickets as resources and have a module which will allocate, monitor the careful transfer and deallocate tickets in the system. Such a *ticket manager* will function like a memory or file manager and will be added in the layer 3 of the MINIX system. However this involves handling of low level Solaris details, such as process management and signals. This is a very difficult task and requires an in depth knowledge of the Solaris system.

A Trace of `settickets()`

Following is a trace of the `settickets()` system call. A part of the trace follows from the trace of the `getpid()` system call provided to us by Paul Ashton.

In layer 4:

1) `settickets()` in /src/lib/sunsyscall/settickets.s

- Just branches to `_settickets`

2) `_settickets` in /src/lib/posix/_settickets.c

- Calls `_syscall` passing MM and SETTICKETS as destination and syscall number, and an initialized message struct.

3) `_syscall` in /src/lib/other/syscall.c

- Puts syscall number in message, then passes MM and message to `_sendrec`.

4) `_sendrec` in /src/lib/sun4/sndrec.s

- Loads pid into a register. - Puts call type (send/rcv), destination (MM) and the address of the message on the stack. - Makes SunOS call to block IO and ALRM signals.

5) SunOS in /src/lib/sun4/SunOS.s

- sets up parameters, then traps to Solaris to carry out the syscall

4a) Back in `_sendrec`

- Makes SunOS call (as above) to send signal USR1 to the process. This switches execution into a signal handler in layer 1 ....

In layer 1:

1) SunOSsig in /src/kernel/mpx.c

- Calls `entering_kernel`

2) `entering_kernel` in /src/kernel/sunprotect.c

- Calls `unmap_process` for the user process invoking settickets.

3) `unmap_process` (same file)

- Makes three SunOS calls (all as above) to remove memory mappings for process settickets called in. - Returns to `entering_kernel` which returns to ...

1b) SunOSsig

- context of user process saved in its smx proc structure by a call to memcpy

2) `memcpy` (/src/lib/ansi/memcpy.c)

- copies the proc structure one character at a time. - returns to ...

1c) SunOSsig

- Call made through the vectors array to s_call (the entry for the USR1 signal).

2) `s_call` in /src/kernel/mpx.c

- calls umap to determine physical address of syscall parameters placed on stack by _sendrec

3) `umap` in /src/kernel/system.c

- returns physical address of syscall parameters back to

2a) `s_call`

- Calls `phys_copy` to copy in the three parameters on the stack.

3) `phys_copy` in /src/kernel/sunprotect.c

- Calls `set_protect` to make the stack area readable.

4) `set_protect` (same file)

- Returns because not full protection. Returns to ...

3a) `phys_copy`

- Calls `set_protect` to ensure kernel array writable (same as above) - Calls `real_phys_copy` to actually copy the bytes

4) `real_phys_copy` in /src/kernel/copySUN.s

- In conjunction with pcbytes, makes the copy. Returns to

3b) `phys_copy`

- Calls `set_protect` (as above) twice to restore protection. - Returns 2 ...

2b) `s_call`

- Calls `sys_call` with the parameters copied from the stack (BOTH, MM, message_ptr into user address space).

3) `sys_call` in /src/kernel/proc.c

- Does some validation—everything checks out. - Call is a send/rec, so calls `mini_send`

4) `mini_send` (same file)

- Does several validation checks—everything checks out. - The destination (MM) is waiting to received this message, so cp_mess is called to transfer the message.

5) `cp_mess` (same file), SHADOWING is 0, logging not enabled

- uses `umap` (see above) to compute source and destination physical addreses. - uses `phys_copy` (see above) to copy the message from the user process into the kernel. - the message source is sent to the sending process number - uses `phys_copy` (see above) to copy the message from the kernel to MM. - returns to ...

4a) `mini_send`

- MM is marked as no longer receiving, and return value for MM system call is set to OK. - MM `p_flags` now 0 so ready is called to put MM back on the ready list.

5) `ready` (same file)

- the server queue is empty, so MM is placed at the head of the server queue. - returns to `mini_send`, which returns to

3a) `sys_call`

- function is BOTH, so `mini_rec` is called for the user process.

4) `mini_rec` (same file)

- no messages are waiting, and no blocked interrupts are waiting - `event_log` is called

to record an EV_MSG_BLOCK for the user process

5) `event_log` (/src/kernel/logging.c)

- logging is disabled, so it returns immediately to ...

4a) `mini_rec`

- proc entries set to record who message is expected from, and location of message buffer. - `unready` is called to remove the process from the ready list

5) `unready` (/src/kernel/proc.c)

- `rdy_head` for user is set to nextproc (NIL_PROC in this case). - `pick_proc` called to choose a new process to run.

6) `pick_proc` (same file)

- the first process found is MM in the server queue. `proc_ptr` is set to MM, and pick_proc returns to unready, which returns to

4b) `mini_rec`

- user process marked as RECEIVING - MM has not just blocked, so we return to `sys_call` which returns to ..

2c) `s_call`

- sets return register of user process with the result of `sys_call`, then returns to ...

1d) SunOSsig

- calls `lock_pick_proc`

2) `lock_pick_proc` (/src/kernel/proc.c)

- sets switching, calls `pick_proc` (as before chooses MM), resets switching - returns to ...

1e) SunOSsig

- calls `resume`

2) `resume` (/src/kernel/mpx.c)

- calls `memcpy` (as above) to copy context structure from MM's proc table entry to the layer 1 stack. - calls `leaving_kernel`

3) `leaving_kernel` (/src/kernel/sunprotect.c)

- is not a user process, so no need to map it (MM always mapped). - protection is not full, so simply returns to ...

2a) `resume`

- uses SunOS (as discussed above) to switch execution to MM in layer 3

In layer 3:

3a) `_receive` in /src/lib/sun4/sndrec.s

- execution resumes after the call to kill(2). - sigprocmask(2) is called via SunOS (described above) to restore an empty signal mask. - Then we return to ...

2a) `get_work` in /src/mm/main.c

- sets the globals who (to the layer 4 process) and `mm_call` (to SETTICKETS) - then returns to ...

1a) `main` (same file)

- does some initialization and validation, then calls `do_settickets` via the `call_vec` function pointer array.

2a) `do_settickets` in /src/mm/forkexit.c

- executes `do_settickets` which searces for the process with process id as destpid and sets `mp_proctick` to the value of newtick which is the number of tickets assigned to the process as passed by the user process. - makes a call to `sys_settickets` in layer 2.

In layer 2:

1) In file /src/lib/syslib/sys_settickets.c

- sets the fields of the message to the pid and the number of tickets to be set and then makes a call to `_taskcall`. The destination task for the process is SYSTASK.

2) In file /src/lib/syslib/taskcall.c

- The implementation of `_taskcall()` is the same as `_syscall()` and as before it identifies the receiver of the message and passes the message. The message also contains a field which is set to the syscall number

`_sendrec` executes exactly the same way before except now the process waiting to receive the message is the SYSTASK rather than the MM.

In layer 2:

3) `sys_task()` in file /src/kernel/system.c

- `sys_task()` is implemented and the message is received - type of the message is recognized and the `do_settickets()` routine is called

4) `do_settickets` in the same file

- This routine searches for the process in the process table with process id found in the message field. - Then it sets the p_proctick field of this process with the number of tickets specified in the second field of the message. - Successful execution of this routine sets the value of r to OK and control returns back to..

5) `sys_task()` in the same file

- This routine will now return the value of the call to the sender of the message which in this case is MM.

After execution of send as before control returns back to MM in layer3

Back in layer 3

2b) `do_settickets` returns the code OK.

1b) `main`

- calls `reply` passing return code as OK.

2) `reply` in /src/mm/main.c

- does some validation checks - puts return code OK in the `mm_out` message. - calls send

to send `mm_out` back to the user mode process

3) `_send` in /src/lib/sun4/sndrec.s

- Proceeds as for `sndrec` above except that the parameter setup is somewhat different (destination is layer 4 process; operation is SEND). - The USR1 signal switches execution back into ...

Layer 1 in SunOSsig

Execution is pretty much the same as before.

- `unmap` returns immediately because the process switched from (MM) is not user process.

- `sys_call` called with parameters (SEND, user process, reply msg).

- `mini_send` copies msg from MM to user process; user process is added to the layer 4 ready list queue by `mini_send` - `mini_rec` not called because this is a SEND.

- `lock_pick_proc` chooses MM as it is higher priority than the other ready process (the user process).

- `resume` returns execution to MM ...

In Layer 3 (MM):

3a) `_send`

- execution resumes after the call to kill(2). - `sigprocmask(2)` is called via SunOS (described above) to restore an empty signal mask. - Then we return to ...

2a) `reply` in /src/mm/main.c

- checks return result of `_send` OK, then returns to ...

1c) `main` (same file)

- returns to top of look and calls `get_work`

2) `get_work` (same file)

- Calls `_receive`

3) `_receive` in /src/lib/sun4/sndrec.s

- Proceeds as for `sndrec` above except that the parameter setup is somewhat different (src is ANY; operation is RECEIVE). - The USR1 signal switches execution back into ...

Layer 1 in SunOSsig

Execution is pretty much the same as the last call from MM.

- `sys_call` called with parameters (RECEIVE, ANY, msg buffer).

- `mini_send` not called by `syscall` because this is a RECEIVE

- `mini_rec` is called operating much as in the first system call. unready suspends MM and `pick_proc` chooses the user process (it is the only one ready). MM has just blocked, but `sig_procs` is 0 so inform is not called.

- `lock_pick_proc` also chooses the user process.

- In `leaving_kernel` the process being resumed is a user process, so `map_process` is called

4) `map_process` in /src/kernel/sunprotect.c

- No currently mapped process, so `unmap_process` not called. - Makes three SunOS calls (all as above) to add memory mappings for the three segments of the user process `settickets` called in. - Returns to `leaving_kernel` which returns to `resume`.

- `resume` returns execution to the user process ...

In user process

4b) `_sndrec` - execution resumes after the call to kill(2). - sigprocmask(2) is called via SunOS (described above) to restore an empty signal mask. - Then we return to ...

3a) `_syscall`

- does some validation. - returns `m_type` field of the message sent by MM to ...

2a. `_settickets`

- returns the value returned by `_syscall`, which is OK returned from MM

B Changes in the code

Following is a listing of the files to which changes are made :

1. **/include/minix/com.h** - This file contains the various task numbers, function codes and the reply codes. `SYSTASK` has a task number of -2. The different internal functions for this task have functions codes assigned from 1-20. `settickets` being an additional function call the function code for this is assigned to be 21.

2. **/include/minix/syslib.h** - This file contains the prototypes for the system library functions. A prototype for the `settickets()` function is added.

3. **/include/minix/callnr.h** - This file defines the system call numbers. The new system call is added to this table. The new system call is defined to have a system call number 50. This is an unused call number and hence used to add the extra system call.

4. **/include/sys/types.h** - This file contains the data types used in the different structures, inode etc. The data type for tickets is defined to be `int`.

5. **/include/unistd.h** - This files contains the prototype for the system call `settickets()`.

6. **/src/lib/posix/_settickets.c** - This file contains code for implementing the message passing between the user level to the memory manager ie. from layer 4 to layer 3

7. **/src/lib/syslib/sys_settickets.c** - This file contains the code for message passing between the memory manager and the kernel ie. from layer 3 to layer 2.

8. **/src/lib/sunsyscall/settickets.s** - This file contains the Solaris related code. It contains a branch instruction to be branch to `_settickets` in */src/lib/posix/_settickets.c*

9. **/src/mm/table.c** - This file contains a mapping of the system call numbers and the routines that perform these system calls. The routine which implements the system call `settickets()` is `do_settickets` and is found in */src/mm/forkexit.c*

10. **/src/mm/proto.h** - This file contains the prototype declaration for the routines which are found in */src/mm*. The prototype for the routine `do_settickets` is defined.

11. **/src/mm/mproc.h** - This file contains the declaration for the process entry. Every process entry contains the memory related information and basically represents a slot in the process table. Thus for a process this slot number is the same slot number for the kernel as well as the file system. In addition to the other information an extra field is now added that contains information about the number of tickets for every process.

12. **/src/mm/forkexit.c** - This file contains the implementation for the `settickets()` system call. The routine basically looks for the process with the process identifier `destpid`. It searches in the process table for the correct process entry. It then modifies the value of the tickets assigned to this process with the new number of tickets specified by the user as `newticks`. Then a call to the system task is made which will handle the modification of the ticket information for the kernel.

13. **/src/kernel/proc.h** - This file contains the declaration for the process table. A new field `p_proctick` is added to the process entry. This will contain information about the number of tickets assigned to the process.

14. **/src/kernel/system.c** - This file contains the implementation of the system call. This system task handles all these system calls. The system task receives the message and identifies the routine to be called. The `do_settickets` routine will implement

the system call `settickets()`. `do_settickets()` is passed a message pointer. It will again look in the process table for the process with the pid which is passed as a field in the message. It will then modify the ticket count for that process with the new ticket count again specified in the second field of the message. If system call is implemented correctly it returns a code `OK`.

References

[Ash96] Paul Ashton. Smx-the solaris port of minix. Technical report, September 1996.

[Com82] Houghton Mifflin Company. *The American Heritage Dictionary*, page 1097. Second college edition edition, 1982.

[Mil97] Milan Milenkovic. *Operating Systems Concepts and Design.* Tata McGraw Hill, second edition, 1997.

[PMG] David Petrou, John W. Milford, and Garth A. Gibson. Implementing lottery scheduling: Matching the specializations in traditional schedulers. pages 1–14.

[SHS99] D. Sullivan, R. Haas, and M. Seltzer. Tickets and currencies revisited: Extensions to multi-resource lottery scheduling, 1999.

[SJP91] A. Silberschatz, J.Peterson, and P.Galvin. *Operating System Concepts.* Addison Wesley, third edition, 1991.

[SS99] Brandon C.S. Sanders and Nathan R. Sprague. Lottery scheduling for the linux 2.2.x kernel. http://www.cs.rochester.edu/u/sanders/linux-scheduler-proj/lottery_scheduler/lottery_scheduler.html, 1999.

[TW97] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*, chapter Processes. Prentice Hall, second edition, 1997.

[Wal95] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management.* PhD thesis, Massachusetts Institute of Technology, 1995.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share reosource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation, Usenix Association*, 1994.