

Lottery Scheduler for the Linux 2.6 Kernel

Maria Helena Mejia Salazar Tapasya Patki
Department of Computer Science
University of Arizona
December 2010

Abstract

This report describes the design and implementation of Lottery Scheduling, a proportional-share resource management algorithm, on the Linux 2.6 kernel. A new lottery scheduling class was added to the kernel and was placed between the real-time and the fair scheduling class in the hierarchy of scheduler modules. We evaluated our scheduler on compute-intensive, I/O-intensive and mixed workloads, and our results indicate that the process scheduler is probabilistically fair and prevents starvation. We also conclude that the overhead of our implementation is roughly linear in the number of runnable processes.

1 Introduction

Process management is one of the major responsibilities of any operating system. This involves allocating various resources to processes. The CPU is the most important resource that must be shared efficiently among many competing processes in a given system. There are plenty of algorithms in the literature that provide efficient and fair scheduling for a CPU. These algorithms focus on maximizing CPU utilization, ensuring fairness to all runnable processes, and improving throughput. Some of these algorithms are First-In-First-Out (FIFO), Round Robin, and Fixed-Priority Preemptive Scheduling.

In 1994, Waldspurger et al. [1, 2] proposed the Lottery Scheduler, a randomized resource allocation algorithm that efficiently implements proportional-share resource management. This scheduling algorithm solves the problems of starvation (perpetual denial of CPU time-slice) and priority inversion, and is also probabilistically fair; unlike other process scheduling algorithms which do not ensure fairness.

We implemented the lottery scheduling algorithm on the Linux 2.6.24 kernel, and this report summarizes our implementation details and results. The rest of this report has been organized as follows. Section 2 provides an overview of the lottery scheduling algorithm. Section 3 discusses our implementation in detail. This section includes our experience with User-Mode Linux, the implementation of the lottery scheduling class, and the integration of the scheduler with the Linux kernel. Section 4 presents an overview of the test cases and test scenarios, followed by a detailed discussion on the results that we obtained. We then conclude our report by stating a summary and future work.

2 Lottery Scheduling

Lottery scheduling is a probabilistic process scheduling algorithm. Each process is assigned a few lottery tickets, and the scheduler holds a lottery to draw a random ticket. The CPU is granted to the process with the winning ticket. The ticket distribution could be non-uniform. If a process has more tickets than other processes, it has a higher probability of winning and for being selected for execution. Since a process with at least one ticket will eventually win a lottery, the problem of starvation is solved and probabilistic fairness is ensured. The throughput of the system as well as the responsiveness depends on the distribution and allocation algorithm used [1, 2].

The lottery scheduling approach can be used for any other kind of resource as well, and not just the CPU. In such situations, we refer to the competing entities as *clients*. We now discuss some techniques for implementing resource management policies with lottery tickets.

Ticket Transfers

This technique is used in situations where a client is blocked holding a resource due to some dependencies, and another client is waiting for the first client to release the shared resource. The idea here is to transfer the tickets to the blocked client, so that it has a higher probability of executing and can finish the task faster and release the resource sooner and both clients benefit from this. A client must have the ability to transfer its tickets to one or more clients that it depends on.

Ticket Inflation

This is considered as an alternative to explicit ticket transfers where a client can create more tickets for itself. On one hand, this approach can present problems in the sense that a client can monopolize a resource by creating a large number of lottery tickets. On the other hand, the inflation/deflation can be used to adjust resource allocations without any explicit communications between clients, leading to a better throughput.

Compensation Tickets

If a client uses only a fraction f of its allocated resource unit (for example, CPU cycles/time), it can be assigned a compensation that inflates its number of tickets by a factor of $1/f$. This ensures fairness and better responsiveness in case of process scheduling. Interactive processes depict this behavior frequently because they enter the wait state while reading data from the disk/memory. Compensation helps these processes get their fair share of the CPU.

Figure 1 shows an example of how this algorithm works. For selecting the winning process, the algorithm generates a random number between one and the total number of tickets in the system. It then traverses the run-queue by accumulating the number of tickets it has seen so far. For each process in the queue, it checks if the accumulated number of tickets is greater than the random number. If yes, then this would be the winning process which holds the randomly drawn lottery ticket. In Figure 1, the total number of tickets in the system is 40 and the random number is 35. For each process, we check whether the number of tickets accumulated so far is greater than 35. We continue until we reach the fourth process, which is declared to be the winner and is granted the CPU.

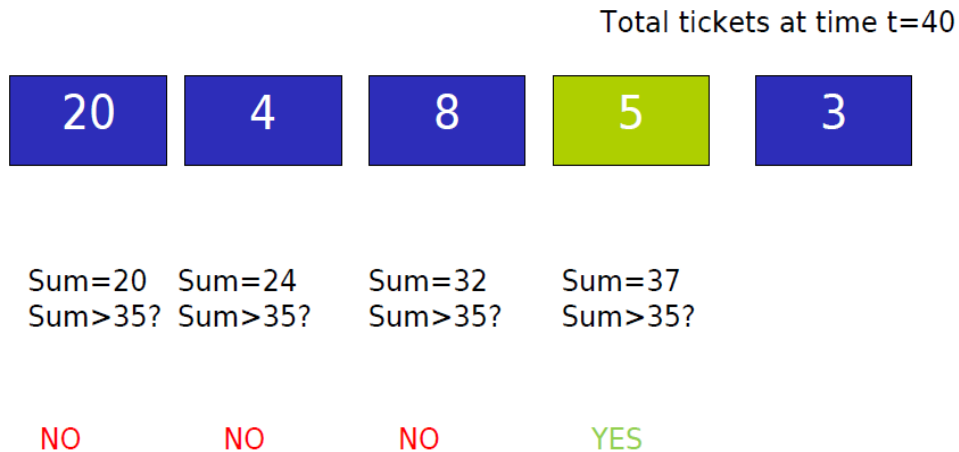


Fig. 1: Lottery Scheduling

3 Implementation

In this section, we present an overview of User-mode Linux and the existing Linux 2.6 scheduler followed by a detailed description of our implementation.

3.1 User-mode Linux

User-Mode Linux (UML) is a virtual machine environment that lets us run Linux guest systems in user-space. It is used for multiple purposes, such as kernel development, testing new distributions and patches, hosting of virtual servers, and running network services independently of a real system. UML was originally designed for the x86 architecture, but can now support other architectures like IA-64 and PowerPC. UML is available under the General Public License (GPL) and has been integrated into the Linux Kernel.

Instead of interacting directly with the hardware, the UML kernel interacts with a Linux kernel like a user-space program. In addition, being a virtual machine, it allows the user to run programs on top of it as if they were running under a normal kernel. Even if UML crashes, the host kernel is still working and is unaffected by the error. This makes UML an ideal platform for kernel development. It is possible to modify and program an instance of a kernel without suffering any damage in the host Linux. UML can also be debugged like any other normal process using standard debuggers like `gdb`. In other words, it is possible to carry out new development and testing at the kernel level in a safe way.

Despite its advantages, UML is limited and does not offer all the functionality that a host Linux system would offer. For instance, it does not support multiple virtual consoles or terminals that are important when testing the kernel, and it is very difficult to set up networking. Another limitation is that testing for multiple users is not feasible. Because of these shortcomings, we could not test our implementation in a multiuser setup as UML did not return to the shell prompt after setting a user, rendering it impossible to set another user. Also, we had to run our processes in the background, as we had access to only one terminal screen and we could not launch multiple processes in the foreground and test them.

UML can be downloaded and built from source. This can be achieved by using the `ARCH=um` option while configuring and building the source. Some features of UML may require a special user-space program, and the `uml_utilities` package can be installed for this purpose. A detailed guide for installing UML can be found here [7].

3.2 Overview of the Linux 2.6 Scheduler

The Linux 2.6 kernel introduced an $O(1)$ process scheduling algorithm that was independent of the number of runnable tasks in the system. The previous scheduler was an $O(n)$ scheduler which was slow and lacked scalability. The pre-2.6 scheduler also used a single run-queue for all the processors and used a single run-queue lock which further degraded performance [3].

One of the main differences in the Linux 2.6 scheduler is that each processor now has its own run-queue, which helps in reducing lock contention. Additionally, the concept of a priority array is introduced which uses the active array and the expired array to keep track of tasks in the system. The $O(1)$ running time is obtained with the help of these new data structures. The scheduler puts all processes that used their time-slice in the current scheduling run into the expired array. When there are no active processes left in active array, it swaps active array with expired array. The scheduler always schedules the task with the highest priority first, and if multiple tasks exist at the same priority level, they are scheduled in a round robin way. The scheduler also achieves load balancing by migrating tasks from busy processors to idle processes. The 2.6 kernel supports `SCHED_FIFO` and `SCHED_RR` for real-time scheduling, and the `SCHED_NORMAL` uses the $O(1)$ policy that we just discussed. This is also known as the Completely Fair Scheduler (CFS).

Linux 2.6 kernel also introduced the notion of scheduling classes, and a hierarchy of scheduler modules.

Each scheduling class encapsulates a scheduling policy. These classes are maintained as a linked list, and provide an extensible framework. The current implementation includes the CFS class, and the RT (Real Time) class which comprises of the `SCHED_FIFO` and `SCHED_RR` policies. When the policy of a task is set to `SCHED_NORMAL` or `SCHED_IDLE`, the CFS scheduling policy is used. This concept of scheduling classes led to a significant refactoring of the process scheduling code. The new `sched.c` code base has a better design, is general, extensible and relatively easy to maintain and understand.

We now provide a brief overview of the important data structures and the `schedule()` function from the Linux 2.6 scheduler. We discuss the fields that are most relevant to our implementation. The next subsection then illustrates how our code integrates with the data structures discussed here.

struct task_struct

This is the Process Control Block. Every process in the system is represented by a `task_struct`. This is a very large data structure that holds together all information related to a process. When a process or a thread is created, the kernel allocates a new `task_struct` for it. The kernel then stores this in a circular linked list called `task_list`. The most important fields from this structure from the point of view of implementing a scheduling algorithm are:

- `state`: It describes the current state of a process, which could correspond to `TASK_RUNNING`, `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, `TASK_ZOMBIE` or `TASK_STOPPED`
- `policy`: holds a value of scheduling policies
- `sched_class`: a pointer that points to schedule class

struct rq

The kernel creates a run-queue of type `struct rq` for each available processor at boot time. This structure further contains a run-queue for each scheduling policy and holds the following information:

- `nr_running`: number of runnable tasks on the run-queue
- `nr_switches`: number of context switches
- `cfs`: run-queue for the CFS scheduling policy
- `rt`: run-queue for the real-time scheduling policy
- `curr`: pointer to the currently executing task
- `lock`: spin lock for the run-queue
- `active`: the active priority array, contains tasks that have not used up their time-slices
- `expired`: the expired priority array, contains tasks that have used up their time-slices

struct sched_class

This structure provides the framework to implement a new scheduling class. It uses the following callback functions. To implement a new scheduling policy, we need to define a new scheduling class. A run-queue has to be created, and queue operations like enqueue, dequeue, requeue need to be implemented. These are discussed below.

- `enqueue_task`: called when a task enters a runnable state; increments `nr_running`
- `dequeue_task`: called when a task needs to go to the wait state; decrements `nr_running`
- `requeue_task`: called when the time-slice of a task expires and it needs to be requeued
- `check_preempt_curr`: checks if a task that entered the runnable state should preempt the currently executing task
- `pick_next_task`: chooses the task that should run next from the run-queue
- `next`: This is a pointer to the next scheduling class in the hierarchy

The `schedule()` function

The `schedule()` function is the most important function in the `sched.c` file. It is in-charge of selecting the next process based upon the scheduling classes and for performing a context switch between the currently executing process and the process that has been selected to execute. This function is called when a timer-interrupt occurs (`scheduler_tick`), when a process wants to sleep or when a process voluntarily yields the CPU (`sys_sched_yield`). The call graph in Figure 2 gives an overview of this process.

The first thing that `schedule()` does is that it disables preemption. It then retrieves the run-queue based on current processor by calling `smp_processor_id()` followed by `cpu_rq()`. Then, it releases the kernel lock on the current task and obtains the lock on the current run-queue. The next step is to invoke the `pre_schedule()` method. At this point, it is time to determine which task should be executed next, and this is done by calling `pick_next_task()`. The scheduler invokes the function based on the scheduling policy, and looks for the appropriate implementation in the corresponding scheduling class. Once the next task has been chosen, the schedule function checks whether the current task is the same as next task and if a context switch is really required. If the two tasks are the same, it simply releases the lock of running queue and executes `post_schedule()`. Otherwise, it performs a context switch and then executes `post_schedule()`.

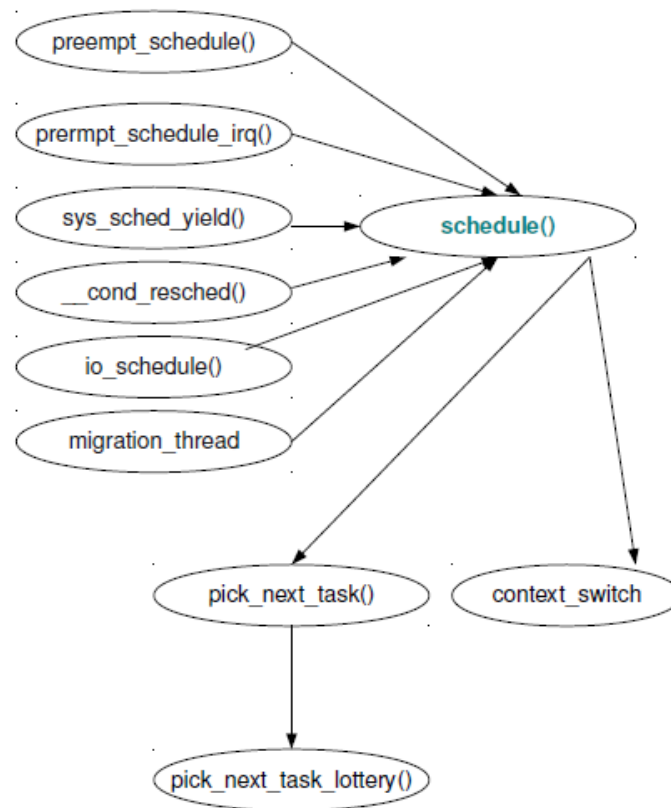


Fig. 2: Call graph for the `schedule()` function

3.3 Implementing the Lottery Scheduler

This subsection describes the detailed implementation of the lottery scheduling policy. We begin by stating our assumptions and the data structures that we introduced in the kernel. We then discuss the scheduling class that we added. Further, Section 3.4 explains our debugging and logging mechanisms [4, 5, 6].

Assumptions

For the scope of this implementation, we make the following assumptions:

- Every new process will be initialized to three tickets
- At any point in time, a process can possess a maximum of five tickets, and a minimum of one ticket.
- Compensation would favor interactive processes and punish processes that have been more luckier than others. If less than 10ms have elapsed since the last time the process was on the CPU, the process loses one ticket. If more than 100ms have elapsed since the last time the process was on the CPU, the process gains one ticket [4, 5].

Initialization and Data Structures

To begin with, a new policy with the name `SCHED_LOTTERY` was added to `include/linux/sched.h` and a configuration option was set up for the lottery scheduler in `arch/um/KConfig`. The `copy_process()` function in `fork.c` was modified to initialize three tickets to each task that was being created in the system. We then modified the process control block structure, `task_struct` and added two fields to this structure that let us account for the number of tickets that the process currently hold, and the previous jiffies value, which is compared against the current jiffies value to check for compensation. A new run-queue that would hold tasks that have their scheduling policy set to `SCHED_LOTTERY` was declared in `kernel/sched.c`. An instance of the lottery run-queue was created in `struct rq`. The following code snippets illustrate these changes.

File: `include/linux/sched.h`

```
#ifndef CONFIG_SCHED_LOTTERY
#define SCHED_LOTTERY    11
#define INIT_TICKETS    3
#define MAX_TICKETS    5
#define MIN_TICKETS    1
#endif

struct task_struct {
...

#ifdef CONFIG_SCHED_LOTTERY
    unsigned int numberOfTickets;
    unsigned long prevJiffies;
#endif

};
```

File: `kernel/sched.c`

```
#ifndef CONFIG_SCHED_LOTTERY
struct lottery_rq
{
    struct list_head lottery_list_head;
    unsigned long nr_running;
};
#endif

struct rq {
...

#ifdef CONFIG_SCHED_LOTTERY
    struct lottery_rq lrq;
#endif

};
```

Implementation of the `lottery_sched_class`

As discussed in Section 3.2, to implement a new scheduling policy, we need to introduce a new scheduling class in the kernel. This involves creating a new run-queue for the scheduling policy, implementing the queue operations and the callback functions in the scheduling class, and making the design decision of where to place this scheduling class in the existing hierarchy of schedulers.

The lottery scheduling algorithm is not real-time, so we decided to place it between the real-time scheduling class and the fair scheduling class. This would mean that any process that is not real-time and has its policy set to `SCHED_LOTTERY` would be given preference over the regular `SCHED_NORMAL` policy, which is the CFS algorithm. This is illustrated in Figure 3.

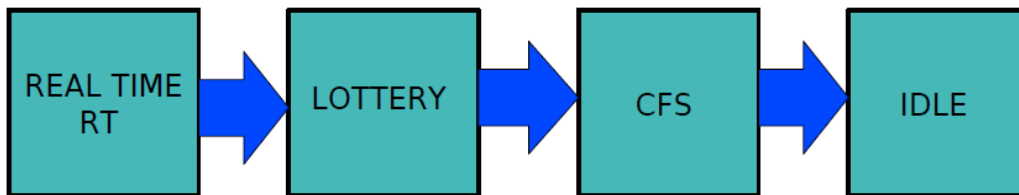


Fig. 3: Hierarchy of scheduler modules

The following code snippet shows the definition of the lottery scheduling class.

```
const struct sched_class lottery_sched_class = {
    .next                = &fair_sched_class,
    .enqueue_task        = enqueue_task_lottery,
    .dequeue_task        = dequeue_task_lottery,
    .yield_task          = yield_task_lottery,

    .check_preempt_curr  = check_preempt_curr_lottery,

    .pick_next_task      = pick_next_task_lottery,
    .put_prev_task       = put_prev_task_lottery,
    .set_curr_task       = set_curr_task_lottery,
    .task_tick           = task_tick_lottery,
};
```

As discussed earlier, the main algorithm for a scheduling policy is contained in `pick_next_task`. For the lottery scheduler, this function uses the following algorithm.

Step 1: Check for compensation

This is done by subtracting the value of `p->prevJiffies` from `jiffies`. The number that we obtain gives us the time elapsed since the last time the process `p` was on the CPU. If this is less than 10ms, the

process loses one ticket. If this is greater than 100ms, the process gains one ticket. While using `jiffies` is fine-grained enough for compensation, we realized that using a high-resolution timer like `RDTSC` could produce more accurate values. We modified our initial implementation to use exact CPU cycles and the processor clock frequency to obtain the elapsed time. A detailed discussion on timers can be found in Section 3.4.

Step 2: Count the total number of tickets in the system

We traverse the run-queue and add the total tickets present in the system at this time. Note that this depends on the number of runnable processes available, and is a linear scan.

Step 3: Draw a lucky ticket randomly

The `get_random_bytes` function is used to draw a random ticket in the range of 1 to total number of tickets in the system (as calculated in Step 2).

Step 4: Search for the winning process

As discussed in Section 2, we now scan the queue and search for the process with the lucky ticket. This is done by accumulating the tickets of the processes we have seen so far in the queue, and then checking whether the current process has the lucky ticket. This check is performed by comparing the accumulated value with the random value obtained in Step 3. This algorithm scans the entire queue in the worst case, and is hence linear in the number of runnable processes.

Step 5: Set next to the winning process, update the `prevJiffies` (or TSC value) and return.

This step returns back the control to the `schedule` function that had invoked `pick_next_task`. The `schedule` function then does a context switch and allocates the CPU to the winning process.

Note: A code snippet of `pick_next_task` has been attached in Appendix A.

3.4 Debugging and Logging Information from the kernel

One of the most challenging issues when writing kernel code is debugging. Kernel code cannot be easily traced or attached to a debugger. The most common debugging technique for application programming is to use `printf` statements. The same can be accomplished in kernel code with the help of the `printk` statement. Priorities, or *log-levels*, can be associated with `printk` statements. These log-levels are defined as macros. Some examples of log-levels include `KERN_INFO`, `KERN_ALERT`, `KERN_ERR`, `KERN_CRIT` and `KERN_DEBUG`. Depending on the log-level, the message could be printed to the console or terminal. If the `klogd` and `syslogd` daemons are running on the system, these messages are appended to `/var/log/messages`. If `klogd` is not running, the message would have to be retrieved by reading the `/proc/kmesg` file. The `printk` function writes messages out to a circular buffer. The `klogd` process retrieves these and dispatches them to `syslogd`, which in turn decides whether or not to print the message to the console based on its priority.

Although `printk` can be successfully used for debugging, it can slow down the system substantially. This is because `syslogd` needs to synchronize all the output file views, which means that every `printk` would incur a disk I/O penalty. `syslogd` tries to write everything out to the disk as soon as possible because the system might encounter a fatal state or could crash after printing out information from the kernel. Because of this limitation, debugging by printing is not considered to be the best practice when writing kernel code. The better approach is to query the system for relevant information when needed, instead of continuously producing data that might not necessarily be read. Thus, it is more appropriate to add a new file to the `/proc/` file system and use the strategy of debugging by querying. We provide a brief overview of how the `/proc/` file system works and discuss our event-logging mechanism in the next subsection.

Adding a Log to the /proc/ File System

The `/proc/` file system is widely used in Linux to obtain statistical and configuration information. It is a pseudo file system that is used by the kernel to export its internal information and state to the user space. Files in the `/proc/` file system do not map to physical files on disks and hence do not take up any space. The contents of the files are generated dynamically when a user attempts to read the file and are stored in memory as opposed to the disk. The `/proc/` file system registers itself with the Virtual File System (VFS), however, when VFS requests it for `i-node` or directory information, it creates these files/directories on the fly from information within the kernel. It thus provides a view into the running kernel.

The `linux/proc_fs.h` header file provides the API that can be used to create `/proc/` entries. `/proc/` entries are created based on the `proc_dir_entry` structure, that links the created entry to the callback functions that would be used when it is read/written to. The following code segment depicts this.

```
File: fs/proc/proc_misc.c

//File operations
static int sch_open(struct inode *inode, struct file *file) { .. }
static int sch_read(char *filp, char *buf, size_t count, loff_t *f_pos) { .. }
static int sch_release(struct inode *inode, struct file *file) { .. }
//Callback functions
static const struct file_operations proc_sch_operations = {
    .open      = sch_open,
    .read      = sch_read,
    .release   = sch_release,
};

struct proc_dir_entry *sch_entry;
sch_entry = create_proc_entry("scheduler_log", 0666, &proc_root);
if (sch_entry){
    sch_entry->proc_fops = &proc_sch_operations;
    sch_entry->data=NULL;
}
```

The next code segment explains how we implemented the logging mechanism in our kernel. Here, `sch_event` is defined as the basic unit of logging in the `/linux/sched.h` file. This structure comprised of an action (like enqueue, dequeue, context switch or debug) that identified a significant event, a time-stamp to record when the event occurred, and a message that could contain some extra information about the event. A structure for holding the log that would record a series of events that occurred is also created. We further defined functions that would help initialize and obtain the log and register an event. The `register_sch_event()` is the key function that gets called whenever an event needs to be logged.

```

File: include/linux/sched.h

#define MSG_SZ          200
#define MAX_EVT_LINES  1000

#define LOTTERY_CONTEXT_SWITCH 3

struct sch_event
{
    int action;          /*Any of the above #defines */
    unsigned long long ts; /*TimeStamp*/
    char msg[MSG_SZ];
};

struct sch_event_log
{
    struct sch_event schEvent[MAX_EVT_LINES];
    unsigned long lines;
    unsigned long cursor;
};

void init_sch_event_log(void);
struct sch_event_log* get_sch_event_log(void);
void register_sch_event(unsigned long long t, char *m, int a);

```

Figure 4 (next page) shows a call graph that explains how this works.

Accurate time-stamping

Another important aspect of this implementation was the resolution of the timers that we used for compensation and time-stamping. The kernel keeps track of time using timer interrupts. These interrupts are generated by the timing hardware, and an architecture-dependent value for this is defined in the `Hz` variable in `linux/param.h`. This value specifies the number of interrupts that would occur per second, and is usually set to 100, or 1000. Every time a timer interrupt occurs, a counter maintained by the kernel called `jiffies` is incremented. This counter must be treated as read-only when writing kernel code. If the `Hz` value is set to 1000, a timer interrupt would occur every millisecond (1000 times in a second), and `jiffies` would be incremented every millisecond. A lower value of the `Hz` variable would mean lesser interrupt handler overhead and slower wrap around; however, this would lead to a low resolution and sampling rate of the kernel time-line.

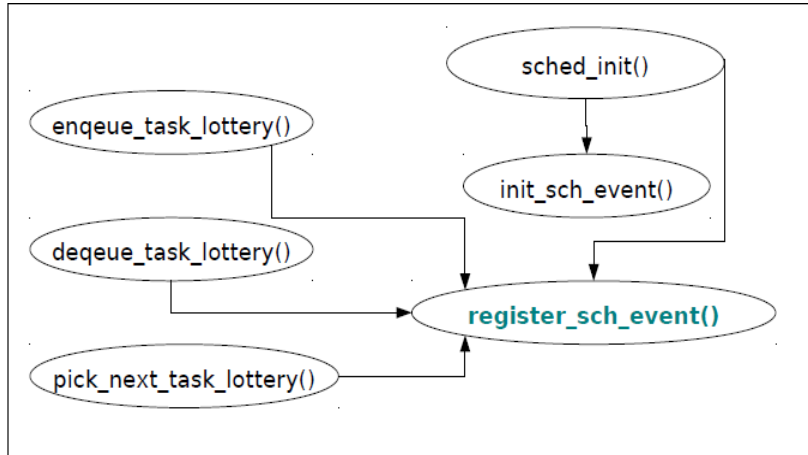


Fig. 4: Call graph for register_sch_event()

Lottery scheduling uses compensation tickets, which means that a process that has not been “lucky” enough to win and has not been scheduled recently needs to be favored by granting it additional tickets. This requires accurate information about the time associated with the process. Also, when measuring scheduling overhead, we want to measure the time taken by the `pick_next_task()` to scan the queue and select the next task to be run. Both these operations, namely, checking for compensation and measurement of overhead need higher resolution timers than `jiffies`. The main reason for higher resolution is that modern processors have a clock cycle of the order of GHz, and this means that perform tasks faster and need to be sampled at a resolution in the range of 0.5 to 1 nanoseconds.

To address this issue, the RDTSC time stamp counter, which is a 64-bit machine specific register available on Pentium machine was used. This counts the number of processor cycles, and is incremented once per clock cycle making it highly accurate. RDTSC can be accessed from user space as well as kernel space with the help of the API present in `asm/msr.h`. This header file provides three functions to read the RDTSC value. These are `rdtsc(low, high)`, `rdtscl(low)` and `rdtscll(long_val)`. The first function reads the lower and higher order bits of the register in the `low` and `high` variables. The second function reads the lower order bits into `low`, and the third function reads the entire 64-bit value in an unsigned `long long` variable `long_val`.

The following code snippet shows the usage of the `rdtscl()` function to gather the lower-order 32-bits of the RDTSC counter. We used this to measure scheduling overhead, and also to calculate compensation accurately by measuring processor cycles.

```

#include <asm/msr.h>
unsigned long rdtsc_start, rdtsc_end, rdtsc_cycles;

rdtscl (rdtsc_start);
//code to be measured
rdtscl (rdtsc_end);

rdtsc_cycles = rdtsc_end - rdtsc_start;
  
```

4 Evaluation

As discussed in Section 1, the goals of a scheduler are to ensure fairness, prevent starvation, provide a good response time for interactive processes and have a high system throughput. In a multi-user system, it is also important to ensure load insulation. Another important aspect to be considered when evaluating a scheduling policy is the overhead incurred by the algorithm. We address these aspects in this section and present our results.

All our experiments were run with root permissions, in a single-user, User-mode Linux setup with SMP disabled. UML was launched with 512MB of memory.

4.1 Test Cases

We used two test-cases to evaluate the performance of our scheduler. The first test-case is compute-bound. This test spawns four processes, and each process runs a long computational loop. To preclude the compiler from optimizing the loops, it is necessary to generate values that cannot be determined by the compiler in advance. Also, the loop termination condition needs to be unpredictable. This ensures that poor cache performance for the loop and that the computations cannot be incrementally calculated. This is important when evaluating fairness, because processes spawned later could take advantage of relevant data being present in the cache, and this would largely bias the results.

The second test-case is I/O-intensive. A structure that comprises of integer, floating-point and string data that is a little bigger than the memory page size is declared. We then write out thousands of such records to the disk, and read these in both random and sequential manner. After each operation, a cache flush is performed to ensure that we have a fair comparison between processes.

4.2 Test Scenarios

Equipped with the test-cases described previously, we evaluated how fair our scheduling policy is. The following test scenarios were considered:

- Compute-intensive workload
- I/O-intensive workload
- Mixed workload, that comprises of both compute-intensive and I/O-intensive processes.

For the compute-intensive workload, we executed up to five instances of our compute-bound test-case. Similarly, for the I/O-intensive workload, we ran up to five instances of our I/O-bound test-case. For the mixed workload, we ran one instance of the compute-bound and two instances of our I/O bound test-case.

Since the lottery scheduler is expected to be linear in the number of processes in the run-queue, it was necessary to evaluate the overhead of our scheduling policy. As discussed in the previous section, an important aspect of measuring this overhead was the accuracy and resolution of our time-stamping mechanism. We used the RDTSC time stamp counter to measure the exact number of CPU cycles taken for the `pick_next_task()` function to count the total tickets in the system, draw a random ticket and scan the queue to find the process holding the winning ticket. All the results have been presented in the next sub-section.

4.3 Results

Figures 5 and 6 depict our fairness results. The first set of graphs shows compute-intensive workload, and the second set of graphs illustrates the I/O-intensive and mixed workloads. When ran individually, the CPU-bound process (just one of the forked processes) takes about 3.5 seconds to execute. When 12 such processes were launched, it was observed that about half of these finished in less than 4 seconds. Two of the processes took as long as 4.6 seconds to execute. Similarly, when twenty such processes were launched, we observed that three of these took more than 4.5 seconds to execute, while most of the processes finished in less than four seconds.

We can conclude from these results that the lottery scheduler is probabilistically fair, and does not starve compute-bound processes. Also, as there is some randomness involved, even with compensation, there may be a few processes (about 10 – 15 % of the processes launched) that may be slowed down significantly.

We ran a similar test for I/O-bound processes, and observed the same phenomenon. When executed in isolation, the I/O bound process takes a little over 0.55 seconds to finish. When five such processes were launched, we observed that two of these finished a little earlier than expected (in about 0.53 seconds), and two other processes with the same workload took about 0.7 seconds.

For the mixed workload, we observed an important trend-- the performance of the I/O-intensive tasks was not influenced by the compute-intensive tasks. This depicts the effectiveness of using compensation tickets. In summary, we observed that our implementation of the lottery scheduler was probabilistically fair, had good performance with I/O-intensive workload, and prevented starvation.

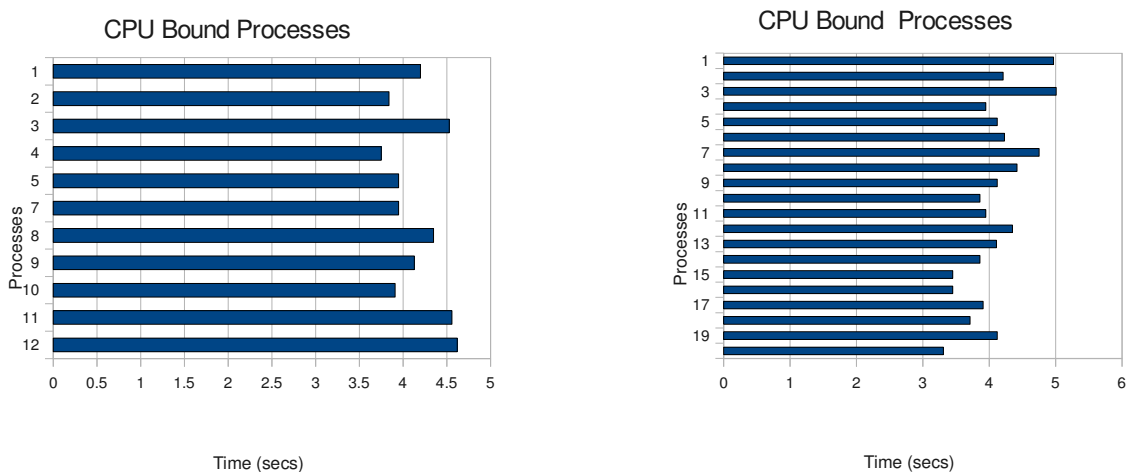


Fig. 5: Fairness of CPU-intensive workload

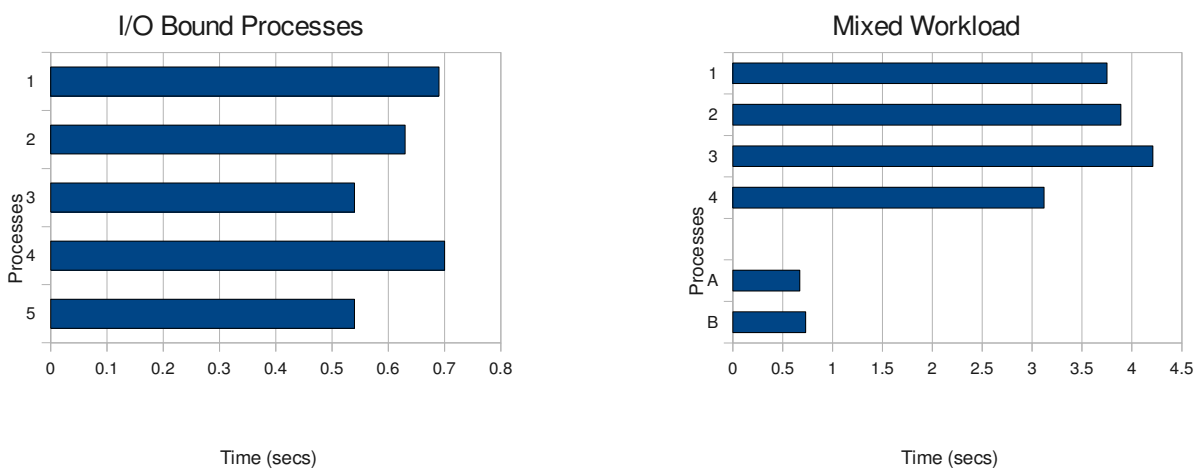


Fig. 6: Fairness for I/O-intensive and mixed workload

Figure 7 presents the overhead of our scheduling policy. Theoretically, we expect the scheduler to take linear time in the number of runnable processes, as we scan the run-queue to determine the process with the winning ticket when we pick the next task. We noted a few anomalies in our actual results though, and we would attribute these to the error in RDTSC measurements because of wrapping around of the counter values. Also, we conducted the experiment on UML and not on a real machine. Another thing to note here is that the overhead includes the scheduling overhead for background processes and daemons that were running in the system, like the `kthreadd`, `ksoftirq`, and `keventd`. We observed that the overhead is still roughly linear, especially after a queue length of six, beyond which most processes are the user-space processes.

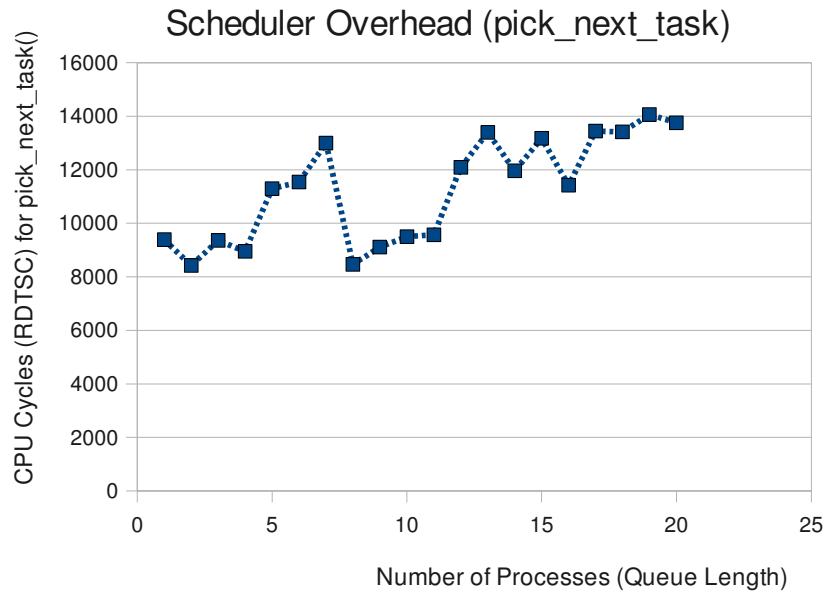


Fig. 7: Overhead of the Lottery Scheduler

5 Conclusions

We presented the design and implementation of the lottery based process scheduling algorithm for the Linux 2.6 kernel. Our results indicate that the scheduling policy is fair, and that compensation tickets help in boosting the performance of I/O-bound processes. Also, the scheduler prevents starvation and priority inversion. We also measured the scheduling overhead and observed that the algorithm is roughly linear in the number of runnable processes, as expected.

Some suggested enhancements would include implementing a $O(\log n)$ algorithm for picking the next task. This would reduce the overhead of the scheduler significantly. Also, as discussed in [1, 4] the performance can be further improved by using kernel priorities and abbreviated quanta.

References

1. C. A. Waldspurger, *Lottery and Stride Scheduling: Flexible proportional-share resource management*, Technical report, Cambridge, MA, USA, 1995
2. C. A. Waldspurger, William Weihl, *Lottery Scheduling: flexible proportional-share resource management*, In OSDI '94, Proceedings of the 1st Usenix conference on Operating Systems Design and

Implementation, Berkeley, USA 1994.

3. <http://oreilly.com/catalog/linuxkernel/chapter/ch10.html>
4. <http://www.hypersurface.net/dpetrou/research.html>
5. http://fcolakoglu.com/lottery_scheduling.html
6. <http://www.cs.rochester.edu/u/sanders/linux-scheduler-proj/>
7. <http://user-mode-linux.sourceforge.net/>

(Note: Appendix A on next page.)

Appendix A

Code snippet for the pick_next_task function. The algorithm was explained in Section 3.3.

```
File: kernel/sched_lottery.c

static struct task_struct *pick_next_task_lottery(struct rq *rq)
{
    // Declare variables, time stamp counters etc.

    // Check for compensation tickets
    temp = queue;
    list_for_each(temp, queue) {
        p = list_entry(temp, struct task_struct, run_list);

        if(((jiffies - p->prevJiffies) < 1 ) && (p->numberOfTickets > MIN_TICKETS))
            p->numberOfTickets--;

        if (((jiffies - p->prevJiffies) > 10) && (p->numberOfTickets < MAX_TICKETS))
            p->numberOfTickets++;
    }

    // Count Total Number of Tickets
    temp = queue;
    list_for_each(temp, queue) {
        p = list_entry(temp, struct task_struct, run_list);
        totalTickets += p->numberOfTickets; }

    // Draw a lucky ticket randomly
    do{
        get_random_bytes(&luckyTicket, sizeof(unsigned long));
        if(totalTickets == 0){ return NULL;}
        luckyTicket = (luckyTicket % totalTickets) + 1;
    }while(luckyTicket < 0);

    // Search for the process with the lucky ticket
    temp = queue;
    list_for_each(temp, queue){
        p = list_entry(temp, struct task_struct, run_list);
        totalSoFar += p->numberOfTickets;
        if (totalSoFar >= luckyTicket ) break; }

    //Set the next process to the process that has the lucky ticket
    next = p;
    return next;
}
```