# Process Scheduling

ECE 5984 Linux Kernel Programming
Spring 2017
Project 3
Group 5

Ho-Ren Chuang, Christopher Jelesnianski, Mincheol Sung

## Algorithm & Implementation

### Description of Algorithm

For this project it was decided to implement "*Lottery Scheduling"* presented by Waldsburger et. al in 1994. We chose this scheduling algorithm because not only is it a good and concise algorithm, but also because it directly relates to our (Chris's) research.

In a nut shell, "*Lottery Scheduling"* is a probabilistically fair scheduling algorithm that provides control over relative execution rates of computations through the use of lottery tickets. Each allocation of a CPU is determined by performing a *lottery*, and the resource is given to the task with winning ticket. The algorithm effectively allocates resources to competing tasks in proportion to the number of tickets that they hold.

It is a neat way of performing scheduling because the concept of lottery tickets abstracts away the hardware while still addressing the quantification of resource rights among other tasks. What also makes this algorithm stand out is that it is dynamic; a task's number of tickets can vary in proportion to the contention for a given resource. When actually performing a lottery, it is known that it is probabilistically fair because the expected allocation of resources (CPU time) is proportional to the number of tickets a task holds. The number of lotteries won by a task has a binomial distribution.

$$p = \frac{t}{T} \qquad E[w] = np \qquad E[n] = \frac{1}{p}$$

The probability $p$ that a task will win a given lottery is equal to $t$, the number of tickets the task is holding divided by $T$, the total number of tickets in the lottery ecosystem. Repeating this lottery $n$ times, the expected number of wins, $E[w]$, is simply $np$. Furthermore, the number of lotteries required for a client's first lottery win has a geometric distribution, with the expected number of lotteries, $E[n]$, that a task must lose before its first win is equal to 1 over $p$. Effectively, inversely proportional to its ticket allocation.

All of the concepts mentioned so far are the foundation for "*Lottery Scheduling"* and do not change during execution. While additional mechanisms were implemented in Waldspurger et. al's work to add a novel dynamic component to *"Lotttery Scheduling"* they are considered out of scope for this project and instead considered as future work.

### Description of Implementation

Implementation was similar to that of CASIO, in that a majority of the same files were modified to achieve our goal of implementing a Lottery Scheduler.

We implemented a base lottery scheduler around the concept of each task having an unsigned long long ticket variable. This variable corresponds with the priority of the task, where a higher ticket value equals higher priority and vice-versa. Therefore, each of our essential structures: struct sched_param, struct lottery_task, and struct task_struct, contain this variable. When a task is run via our validation program, it is assigned a given value.

Not many changes in sched.c were needed. Relatively, the code added registered our lottery scheduling policy as an additionally valid policy within Linux in functions __setscheduler, __sched__setscheduler. sched_init also was appended to initialize the run queue for our lottery scheduler via init_lottery_rq() from the lottery scheduling class file.

Notably, we had to create our own scheduling class file, sched_lottery.c. This file contains our lottery scheduler specific implementation of all of the functions that control tasks on the run queue during runtime. In the first implementation, we decided upon using a linked list data structure to hold tasks. remove_lottery_task and insert_lottery_task were implemented to remove and insert tasks into the run queue, respectively, while enqueue_task_lottery and dequeue_task_lottery used these functions and updated system statistics like the total number of tickets in the ecosystem. The functions who_is_winner is the backbone of our scheduler. It is first in charge of picking a winning lottery ticket via a randomly generated integer (get_random_int) modulo the total number of tickets currently in the ecosystem. It initiated a temporary total to 0 and then traverses the linked list containing *runnable* tasks. At each entry, it checks whether the current total thus far is greater than or equal to the winning ticket number and if so returning the current task; if the current total is less, it moves on to the next element and repeats this process.

For a small number of tasks, a linked-list implementation is acceptable as not many elements need to be traversed to reach the winning task (Worst Case O(n)). However, this is not always the case as many tasks can be simultaneously running on a computer, thus a different data structure should be considered. We thus ported this working implementation to instead rely on a red-black tree structure for better performance as it has a better worst case when dealing with many tasks (Worst Case O(log n)). Since a RB-tree automatically inserts nodes at a given position based upon its value, a different way of reaching the winner task within the RB-tree was needed. In a linked list, simply traversing a linked list, updating the current value, and selecting the winning task once the current value is greater than or equal to the winning ticket value would get us the next task. Now, in a tree structure we need to decide whether we need to go left or right to get to the winning task. For this reason, nodes within the RB-tree now have (represented as acc_ticket), keep track, and update their partial totals as nodes are enqueued and dequeued from the RB-tree.

The traversing mechanism acts as follows:

When a new task is enqueued, insert_lottery_rb_tree is called, creating a new RB-tree node, and then shuffled in the tree using the tasks "ticket" value as the comparator. After the node is inserted, the function recalc_accu_lottery is run to update all nodes accu_ticket partial total values of the run queue. Removing from the RB-tree is simple as rb_erase is performed and the partial total value calculation is re-run.
When it comes to picking the next winner, pick_winner_rb_task is run to properly traverse the RB-tree after a winning ticket is chosen and where accu_ticket comes into play. This function utilizes accu_ticket to save time traversing the RB-tree by checking the left subtree's partial total essentially

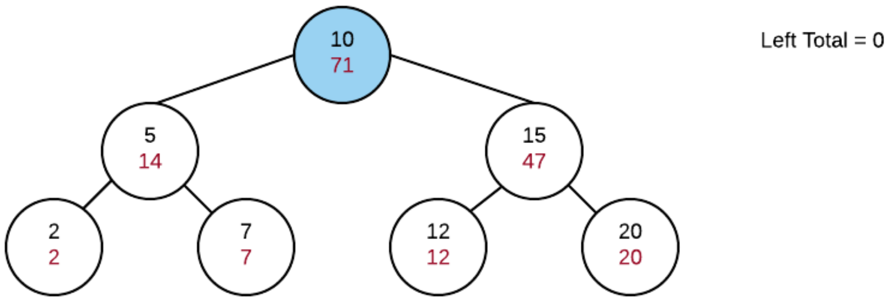
eliminating half of potential candidates to iterate through. So if a winning ticket number is less than the left subnode $\mathrm{accu\_ticket}$, we know the winning task is within the left subtree.

Below we go over a sample traversal of finding the winning task. Node ticket values are in black, Node partial totals are in red.

| | |
|---|---|
| $t_w$ | Winning ticket number |
| $curr\_total$ | Temporary total in this iteration |
| $t_{ac}$ | Left Sub Node's $\mathrm{accu\_ticket}$ value |
| $t_{curr}$ | Current Node's ticket value |
| $left_{total}$ | The current sum of tickets currently to the left and above of the current node |

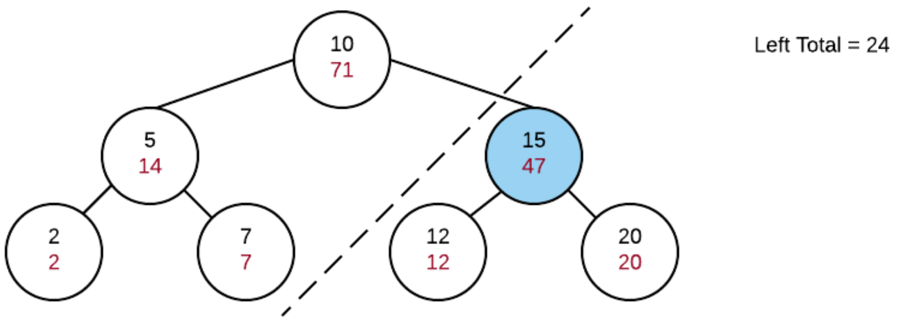
Suppose the winning ticket number, $t_w = 66$. We start at the root node highlighted in blue at A), set $curr\_total = 0$, and first check whether a left sub node exists. If it does, we check if $t_w < t_{ac}$ (It is not, if it were the algorithm would set the current node to the left sub node and restart the process). We update to see if our current node is the winner doing $curr\_total = t_{ac} + t_{curr}$. We check if $t_w \leq curr\_total + left_{total}$. It is not so we can move on to the right sub tree, update $left_{total}$, and repeat the process.

A)



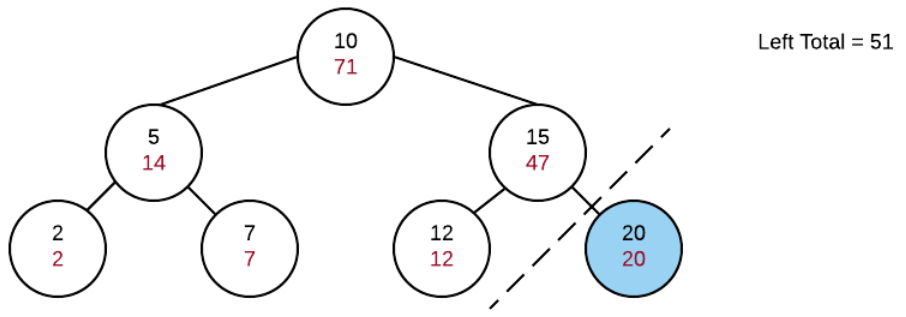
In B)we now see a dashed line representing the cumulative $left_{total} = 24$ from adding $t_{ac}$ (14) to $t_{curr}$ (10). In this new iteration we are performing the same checks as in A) from the current blue node. Yet again we perform the checks to see if we jump to the left subtree (we do not as $t_w$ *is not* $\leq t_{ac} + left_{total}$) or the current node is the winner (it is not as $t_w$ *is not* $\leq curr\_total + left_{total}$). So we move on to the right subtree and update $left_{total}$.

B)



Our $left_{total} = 51$ in C), coming from the previous $left_{total} = 24$ and now adding last iterations left subtree $(15 + t_{ac})$. Again we check if left nodes exist to jump (they don't) or the current node is the winner (It is!! $t_w \leq curr\_total + left_{total}$ $(66 \leq 20 + 51)$). So we have our winner and return the task with 20 tickets.

C)



An interesting problem that came up was when it came time to insert our scheduler among the existing scheduler into the scheduler priority list. Initially we put our scheduler priority at the top (as was done in CASIO) and immediately recorded undesired behavior. Delving into the runtime, it was observed that there was an issue when our user validation program sent signals to tasks to wake them up and let the all the tasks begin running at the same instant. Because the tasks scheduled by our *Lottery Scheduling* have the highest priority on the system (not including the user program who sends the signals), the first woken up task would immediately block the user program from sending any more signals to the rest of the sleeping tasks. This task would be the only available task in the run queue and take up the whole CPU until it is finished. After it would finish, it would dequeue as normal, allowing the blocked user program to wake up and send another signal before being blocked again, repeating this cycle and never allowing more than one task to ever be in the run queue at a time. This effect made the completion order of the tasks always finish in the way they were created (basically FIFO ordering). To solve this bug, we needed to allow our user validation program to be able to interrupt our compute bound tasks and wake the rest of them up to be *schedulable* and on the run queue. To achieve this, we set our validation program to be scheduled as an RT task, while we also downgraded our *Lottery Scheduler* class's priority to be lower than the RT scheduling class. Now the validation program spawning the tasks would no longer be blocked allowing all the tasks to be put onto the run queue.

Finally, to allow each task a fair chance at holding the resource we do implement functionality which takes advantage of time quantum preemption. *Lottery Scheduling* runs a lottery at every task_tick, preempting the current task holding the CPU and giving the new winner access to resources.

Finally, we added functionality inside menuconfig such that you can select "LOTTERY scheduler by using rbtree struct" to use a RB-tree structure instead of a linked list. Our scheduler uses the linked list implementation by default unless this option is selected.

## Validation Program & Results

### Program

For our validation, we decided to focus on and build compute bound tasks as we all focus on High Performance Computing (HPC) in our research. We mostly followed the example available in the CASIO scheduler. The system (lottery_system_fair.c), after reading in tasks with their given number of lottery tickets and amount of time to run, spawning them and immediately putting them to sleep until all the tasks are enqueued, are all launched instantaneously via one signal to wake-up and begin doing their work! The task application (lottery_tasks_fair.c) has been modified to be compute bound and not start until it receives a signal from the main validation program using the “signal” function otherwise falling into an infinite loop containing pause(). To test how different runtimes effect scheduling, we set up a configurable infinite loop (from the input file fed into the validation system program) able to run for a roughly approximated amount of milliseconds.

To debug and gather results about different aspects of the project like fairness, scalability, and other performance data, we implemented printing these information messages to a proc/[1] file, which we then could parse and interpret about our scheduler implementation.

## Plots

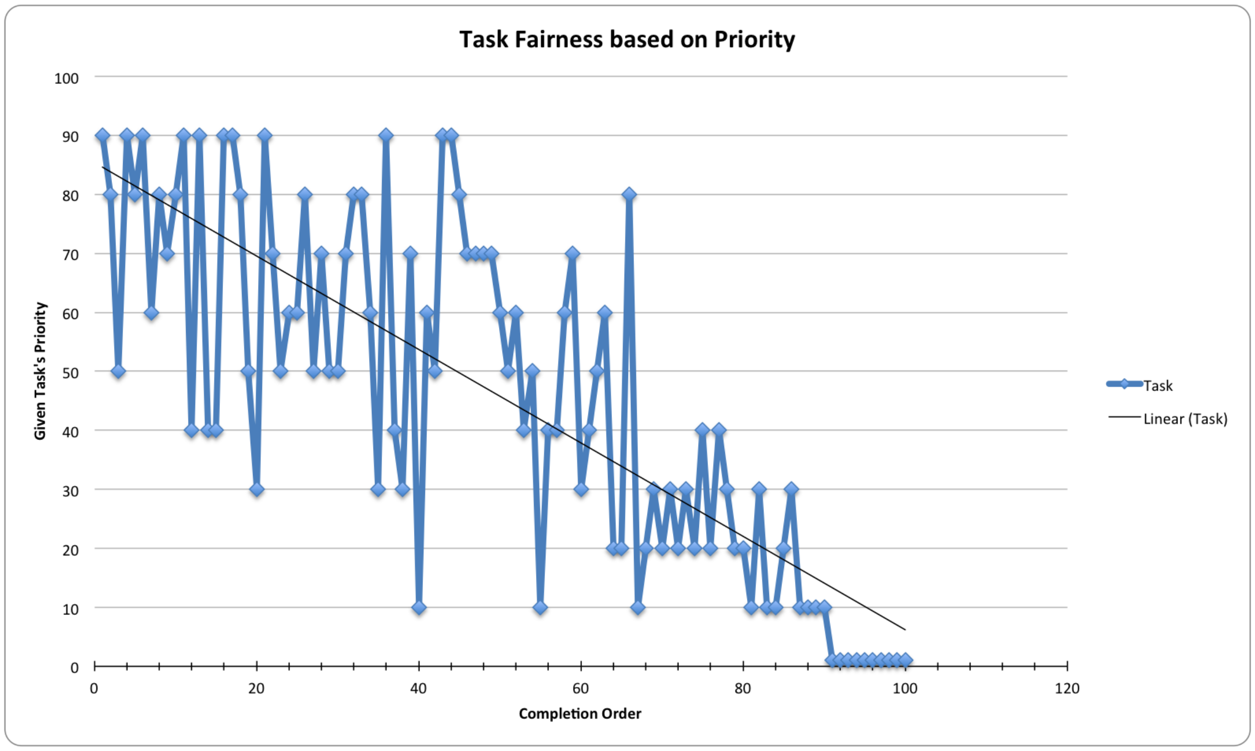


Figure 1

## Comments

It is noted that actual allocated proportions are not guaranteed since the algorithm is based upon a randomized component. Over the course of debugging and recording performance, we observed runs in which low priority tasks sometime finished early than higher priority tasks. Figure 1 above shows just one of the many runs that were performed to evaluate our scheduling policy. This graph plots a task’s priority (on the Y-axis) versus its completion order (on the X-axis) where the first task that completed is

[1] We found that one cannot keep writing proc/ file since eventually it will saturate its buffer. The CASIO sample code doesn't reset the pointer of the buffer, meaning that the buffer is not recyclable. Therefore a solution would be to reset the lines & cursor to the head after read_proc() is done in order to reuse the proc buffer.

the left-most point and the last task is the right most point. In a perfect run, all tasks with the highest priority would finish first followed by the tasks with the next highest priority resulting a graph looking like a descending stair case. Looking at this given run, we see that the tasks follow the intended trend with high priority tasks finished relatively early on (left side) and low priority tasks finish later (right side) on with all of the tasks with the lowest priority finishing last. As is expected, not a perfect staircase shape is observed. From the randomization within the algorithm picking the winner, thus the next scheduled task, while a high priority task has a “better” chance to win, it does not guarantee a win every round.

## Scalability Performance

For obtaining our experimental results, we use the following platform:

| **Machine** | **Core Count** | **Speed** | **Memory** |
|---|---|---|---|
| Host | 8 core | 1.4GHz | 16GB |
| Virtual Machine | 1 core | 4.4GHz | 4GB |

To evaluate the scalability of our scheduling policy, we inserted timer functionality and printed those values to the proc/ file as mentioned earlier. This allowed us to get a per-function breakdown and comparison between the two data structure implementations. We evaluated various task set sizes from 20 tasks up to 3000 tasks in a single run. Again as mentioned earlier, all running tasks were made to be CPU-bound. In addition, we evaluate our validation program in a real world manner. It is unrealistic assuming the run queue size stays constant and saturated (for a designated task count) since tasks always finish and eventually a platform will go back to being idle. Thus, our data includes the time from full saturation of the run queue (for a designated task count) to the time all tasks have finished, therefore also taking into account the time when the run queue is smaller than full saturation. Throughout all our experiments, the task set always uses a 1 to 9 ratio of long running high priority tasks versus short running low priority tasks (e.g., Task set with 10 would have 1 high priority task working 100 units of time and 9 low priority tasks working 1 unit of time). This allows us to evaluate our scheduling policy in a best case real world scenario.

The files used for this performance data are lottery_system.c and lottery_task.c.
Sample Usage is: $sudo ./lottery_system 1000
Where 1000 is the number of tasks to spawn.

For getting decent performance data, we also used *taskset* to pin our VM to work on a dedicated CPU, and *renice* & *chrt* to set the VM to be the highest priority task in the system.

## Responsiveness

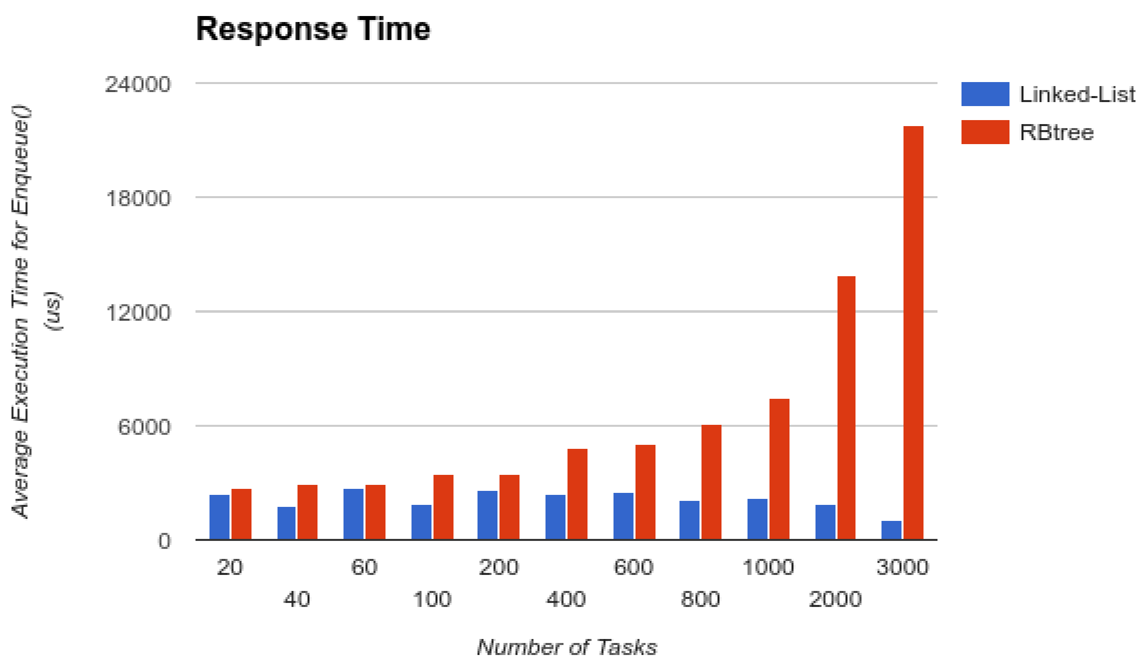


Figure 2

In Figure 2, we observe the average enqueue time needed to insert a new task into our scheduling policy's run queue varying the number of total tasks needed to be run. As can be seen, the RB tree implementation does very poorly when a lot of tasks are in the run queue since RB tree has the need of rebalancing as well as recalculating partial totals after every insertion whereas the linked-list implementation doesn't need to perform these operations minimizing overhead. However, enqueueing is only 1 part of the algorithm.

## Performance

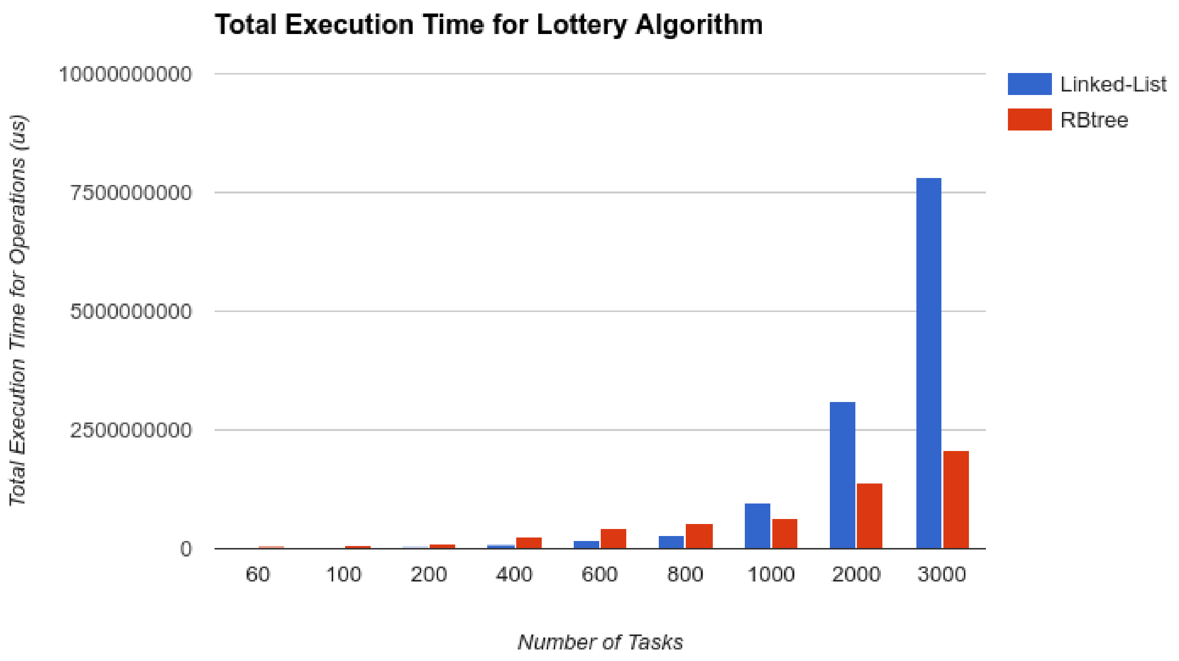


Figure 3

To get a better picture of the actual overall performance of our two lottery scheduling policy implementations, it is needed to also include other vital operations happening in the algorithm. Therefore, we factor in the time needed to enqueue a task, pick a winner and reschedule a task, and dequeue a task when it is completed. For this the code was modified to time the following functions:

- enqueue_task_lottery
- dequeue_task_lottery
- who_is_winner

Figure 3 shows the sum of all three operations total execution time (in us) while varying the number of total tasks in the system for the linked-list implementation in blue and RB tree implementation in red. It turns out that for test cases with small number of tasks, the linked list outperforms our RB tree implementation. Since the run queue is relatively small, the overhead from enqueue takes over. When the number of tasks is increased, this overhead gets increasingly amortized using a RB tree implementation, as the RB tree is able to grab the next task scheduled to run in time complexity O(log(n)) compared to linked-lists time complexity O(n).

In conclusion, our group successfully implemented a static lottery scheduling policy within the 2.6.32.27 kernel using different data structures to compare performance. From these results, it is suggested that a linked-list implementation is adequate in low task count scenarios. If dealing with large task count scenarios a RB tree implementation is optimal as overhead for a linked list handling a large task count sky rockets!

## Future Work

Components (a given processes number of tickets, total number of tickets within the ecosystem, etc.) within these concepts are able to change throughout runtime thanks to Waldspurger's second contribution with the notion of *Modular Resource Management*.
The original publication explores four different techniques for implementing resource management policies with lottery tickets. These being *Ticket Transfers, Ticket Inflation, Ticket Currencies, and Compensation Tickets*. Again, this management is possible because of the abstract-ness lottery tickets possess to represent resource rights.

*Ticket Transfers* are the process of moving tickets from one process to another. This technique can be used in cases of where a given process blocks due to some dependency. Scenarios such as a process waiting for another process to release a shared resource. To allow for progress the dependent process could transfer its tickets to the blocking process. Note that this technique not only allows for one to one transfer, but also has ability for a process to divide its ticket transfers across multiple peers on which it may be waiting on.

*Ticket Inflation* is an alternative to explicit *Ticket Transfers*. Rather, this allows a process to escalate its resource rights by creating more lottery tickets. While this technique could be potentially abused where a process could monopolize a resource by creating a large amount of additional lottery tickets, the ability of having inflation and deflation allows for the adjustment of resource allocations without explicit communication among processes, decreasing overhead, increasing throughput.

An easy way to address a starved process is to simply increase the number of tickets in the ecosystem. In this regard, a few runs should be performed scaling the amount of total tickets in the lottery ecosystem. It is predicted in the original paper, increasing the total number of lottery tickets did improve precision of expected proportions to what was observed. At the other end of the spectrum, it was important to not increase the total number of tickets by a significant amount. While continually increases the total in subsequent runs improved precision, an increased overhead of the lottery algorithm should be observed indicating the need of a fine balance between precision and performance.

Ticket Currencies are a second level of abstraction within Lottery Scheduling to further breakdown priority and having finer granularity control within a group, such as individual threads running within a single process. This technique creates a unique currency for each process, allowing further adjustment of resource allotment between a process's threads. This per-process-currency can then be converted back to the lottery "base" currency, giving each secondary thread a relative resource allotment within the overall lottery.

*Compensation Tickets* is a mechanism that grants compensation to processes that only use a fraction $f$ of its allocated resource by inflating its number of tickets by a factor of $1/f$. This mechanism adjusts a process's resource usage ($f * p$) to its fraction of consumption so that it is still using its' full allocated share $p$.

$$f * p \quad \Rightarrow \quad \frac{1}{f}(f * p) \quad \Longrightarrow \quad p$$

Without compensation tickets, a process that does not use up its entire resource quantum would receive less than its entitled share.