

Lottery and stride scheduling on xv6 with basic statistical analysis

Gabriel Batista Galli¹

¹Federal University of Fronteira Sul (UFFS)
Mailbox 181 – 89.802-112 – Chapecó – SC – Brazil

g7.galli96@gmail.com

Abstract. *This paper describes an implementation of the lottery and stride schedulers in the xv6 operating system. Lottery scheduling is a process scheduling mechanism that randomly chooses a process to run based on its tickets (just like a lottery). The more tickets one process have, the higher the probability of it being chosen by the scheduler. In the code presented here, the Binary Indexed Tree (BIT, or Fenwick Tree) is used to efficiently count every process' amount of tickets. In the stride scheduling, a process still has tickets, but now alongside a stride and a pass. The stride is inversely proportional to its tickets, as it is the ratio between a large constant and the amount of tickets. The pass is used to accumulate the stride and to choose the next process to run: the one with the lowest pass. This way, the tickets serve the purpose of making a process run more often than others, because the higher the quantity of tickets, the lower is the stride. Here, the Segment Tree was used to implement a priority queue in order to efficiently track the changes of passes and find the next process to be executed. Finally, we analyze the scheduler behavior with both mechanisms by looking at what order a group of processes with the same task, but different amounts of tickets, finish their job.*

1. Introduction

The xv6 operating system is a simple Unix-like teaching operating system developed by MIT and widely used in Operating Systems classes all over the world. Originally, every time the scheduler was to choose a process to run, it would linearly search through the array of processes looking for the first one that is `RUNNABLE` [Cox et al. 2014]. In our class, we were assigned to implement the lottery and the stride schedulers on xv6.

As seen in [Waldspurger and Weihl 1994], the lottery scheduler is a probabilistic scheduler algorithm that works similarly to a real world lottery. Every process is given a certain amount of tickets and the scheduler randomly picks one of the available tickets among all processes. The process holding the chosen ticket is then picked to be run by the processor in the next quantum. This way, a process that has a high quantity of tickets has a high probability of being chosen by the scheduler and it will thus be run more frequently than a process that has less tickets.

Being an improvement upon the lottery scheduler, the stride scheduler is a deterministic algorithm [Waldspurger and Weihl. W. 1995]. The concept of tickets is still used here, but every process now has a stride and a pass. After receiving the respective amount of tickets, a stride is assigned to the process, which is the ratio between a large predefined constant (at least equal to or greater than the maximum amount of tickets; 10^5

in this case). The pass is the sum of the stride for every time that process is scheduled by the scheduler. The pass of the processes is used to determine which process among all `RUNNABLE` processes should be given the processor in the next quantum. When we have a tie, we pick the process with the lowest `pid`. This time, being a deterministic algorithm, a process with a high amount of tickets is guaranteed to run more frequently than processes with less tickets.

2. The lottery scheduler

In order to implement the lottery scheduler, a few changes were needed in some data structures used by `xv6` and a new library was created, `lottery.h`:

```
#include "param.h"

#define MAXTICKS      NPROC * NPROC // maximum number of tickets a process is allowed to have
#define MINTICKS      1 // minimum number of tickets a process is allowed to have
#define SYSTICKS      MAXTICKS / 2 // number of tickets for all system processes
#define DEFTICKS      MAXTICKS / 2 // default number of tickets

#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

unsigned long rand(void);
```

The constants defined here are used to control the maximum and minimum amount of tickets any process can hold. Additionally, the `SYSTICKS` is the constant amount of tickets that every system process receives and `DEFTICKS` is for the user if it does not know how many tickets a process should have. The `rand` function was already defined in the `usertests.c` file, but wasn't being used, so it was moved to this library so it can be used when it is time to choose the next process to run.

Then, in the `proc.h` file, the `tickets` attribute was added to the structure `proc`, which describes a process, to hold the quantity of tickets a process has:

```
// Per-process state
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t* pgdir; // Page table
    char *kstack; // Bottom of kernel stack for this process
    enum procstate state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // swtch() here to run process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16]; // Process name (debugging)
    int tickets; // Quantity of tickets assigned to this process
};
```

Some other modifications made in the code relate to the `ptable` structure, found in the `proc.c` file: the `deadstack` array (a stack), its `top` and the `tickets` array:

```
struct {
    int deadstack[NPROC], top;
    int tickets[NPROC + 1];
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

The `deadstack` array is a stack of the currently available `pids`. When `xv6` starts and the function `pinit` is called, `deadstack` will be initialized with all `pids` from `NPROC` to 1 (so the `pid` 1 will be at the top, and the value of `top` will end up being equal to `NPROC`). Then, whenever a new process is created, it will be identified by the `pid` found on top of `deadstack`. Similarly, when a process dies and its position in the `proc` array is once again set to `UNUSED`, the `pid` that was being used by that process is given back to the top of `deadstack` so it can be used again later.

To keep track of every process' tickets, the `tickets` array was created to be used as a Binary Indexed Tree (BIT, also known as Fenwick Tree) to efficiently count how many tickets there are up until a given process (among all existing processes). This way, we can binary search our tree until we find the leftmost position in the array that sums greater than or equal to the ticket that was picked this time around. This position in the BIT is the chosen process `pid` and, because of the `deadstack` way of handling `pids`, it is also its position in the `proc` array (but -1 , because `proc` is indexed from 0 to $NPROC - 1$).

To manipulate the BIT, the functions `uptick` and `ticount` are used. The former updates the BIT whenever a process is created, executed or exits and the latter returns the sum of tickets up until a given position. Just as `deadstack`, `tickets` is initialized in the `pinit` function, where all positions are set to 0.

This data structure was chosen because all operations (either getting the sum or updating it) cost $O(\log n)$ each [Halim and Halim 2013]. As an also $O(\log n)$ binary search is used to find the chosen process, rather than an $O(n)$ linear search, the total resulting complexity ends up being:

$$O(\log n \times \log n) = O(\log^2 n)$$

2.1. Managing `pids`

The attribution of `pids` is done in the `allocproc` function, when creating a new process. As the `top` variable is always one position above the last valid index of the stack, we get it with `--ptable.top`. Likewise, we give a process' `pid` back to the stack in the `wait` function, but this time with `ptable.top++`.

2.2. Updating the BIT

As said before, `uptick` is used to update the BIT whenever a process is created, executed or exits. It is critical that we don't call it any time more or less than needed, as the BIT will end up accumulating a wrong amount of tickets or one process will be considered to have more tickets than it actually does. Also note that passing a positive argument to this function increments the BIT and passing a negative argument decrements it. Incrementing is the act of giving tickets to a process and decrementing is taking them away from it (either permanently or temporarily, to avoid it being picked when it's not `RUNNABLE`).

The BIT is updated in the functions that initialize a process: incrementing on `userinit` and `fork`; when a process is chosen to be executed: decrementing on `scheduler`; when a process gives up the processor because its time is over: incrementing on `yield`; when a process is woken up: incrementing on `wakeup1`, called by

wakeup; and when a process is killed and we have to wake up the parent, if it is sleeping, so it can wait for that process to `exit`: incrementing, for that parent, on `kill` (wakeup is not used here).

2.3. Tickets

As every process now needs tickets to be chosen to run and every process should have an amount of tickets proportional to its importance or priority, the user must be able to tell `xv6` how many tickets its process will need. To accomplish this, the `fork` system call was changed so that it receives that amount of tickets as its sole argument.

Of course, the user might want to cheat and give the highest possible amount of tickets to all of its processes so they run more frequently than everyone else. Unfortunately, we can't do anything but trust in the amount that we receive. The purpose of the lottery is simply run the available processes proportionally to their amount of tickets. There is no way to know how important a user process is. The only treatment done here is to prevent a quantity less than the minimum or more than the maximum (using the `min` and `max` macros). There is also the `DEFTICKS` constant that the user can use, which is also automatically set if we receive 0 as argument.

3. The stride scheduler

Making the stride scheduler work demanded a few more things in the custom library, now named `stride.h`. A `typedef` was also added in the `param.h` file to define the type `ull`, an alias for `unsigned long long int`:

```
#include "param.h"
#include "types.h"

#define MAXTICKS      (NPROC * NPROC) // maximum number of tickets a process is allowed to have
#define MINTICKS      1               // minimum number of tickets a process is allowed to have
#define SYSTICKS      (MAXTICKS / 2) // number of tickets for all system processes
#define DEFTICKS      (MAXTICKS / 2) // default number of tickets
#define MAGIC         100000
#define INF           18446744073709551615ull

#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
#define left(x) ((x) << 1) + 1
#define right(x) (left(x) + 1)

unsigned long rand(void);
```

As you'd expect, the constants refined in the previous `lottery.h` library were kept. In addition to them, we have `MAGIC`, the constant that is used to calculate the stride, and `INF`, the infinity constant ($2^{64} - 1$). The infinity is a value that no `RUNNABLE` process should ever hold as its pass (that is why we increment the pass modulo infinity). All passes are also set to infinity at `pinit` and whenever a process is `RUNNING` or `SLEEPING`.

Similarly, the structure of a process in `proc.h` was also kept, but as said before, variables to hold the stride and the pass were added. Beyond that, it is needed to backup the last valid pass of every process, in the variable `lpass`, so we don't lose it when we set it to infinity, as told in the paragraph above. When a process becomes `RUNNABLE` again, we use `lpass` to update the new valid pass.

```
// Per-process state
struct proc {
```

```

uint sz; // Size of process memory (bytes)
pde_t* pgdir; // Page table
char *kstack; // Bottom of kernel stack for this process
enum procstate state; // Process state
int pid; // Process ID
struct proc *parent; // Parent process
struct trapframe *tf; // Trap frame for current syscall
struct context *context; // switch() here to run process
void *chan; // If non-zero, sleeping on chan
int killed; // If non-zero, have been killed
struct file *ofile[NOFILE]; // Open files
struct inode *cwd; // Current directory
char name[16]; // Process name (debugging)
int tickets; // Quantity of tickets assigned to this process
ull stride; // Ratio between MAGIC and amount of tickets
ull pass; // Current stride accumulated by this process
ull lpass; // Backup of the last valid pass for this process
};

```

Moving forward, the `ptable` structure was modified again. The `deadstack` and the logic of setting and unsetting `pids` introduced with the previous lottery implementation were maintained. But now, instead of the $NPROC + 1$ -sized `tickets` array for the Fenwick Tree, we have the $4 \times NPROC$ -sized `minpass` array, which is the Segment Tree. It is used as a priority queue to efficiently find the next process to run:

```

struct {
    int deadstack[NPROC], top;
    int minpass[4 * NPROC];
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;

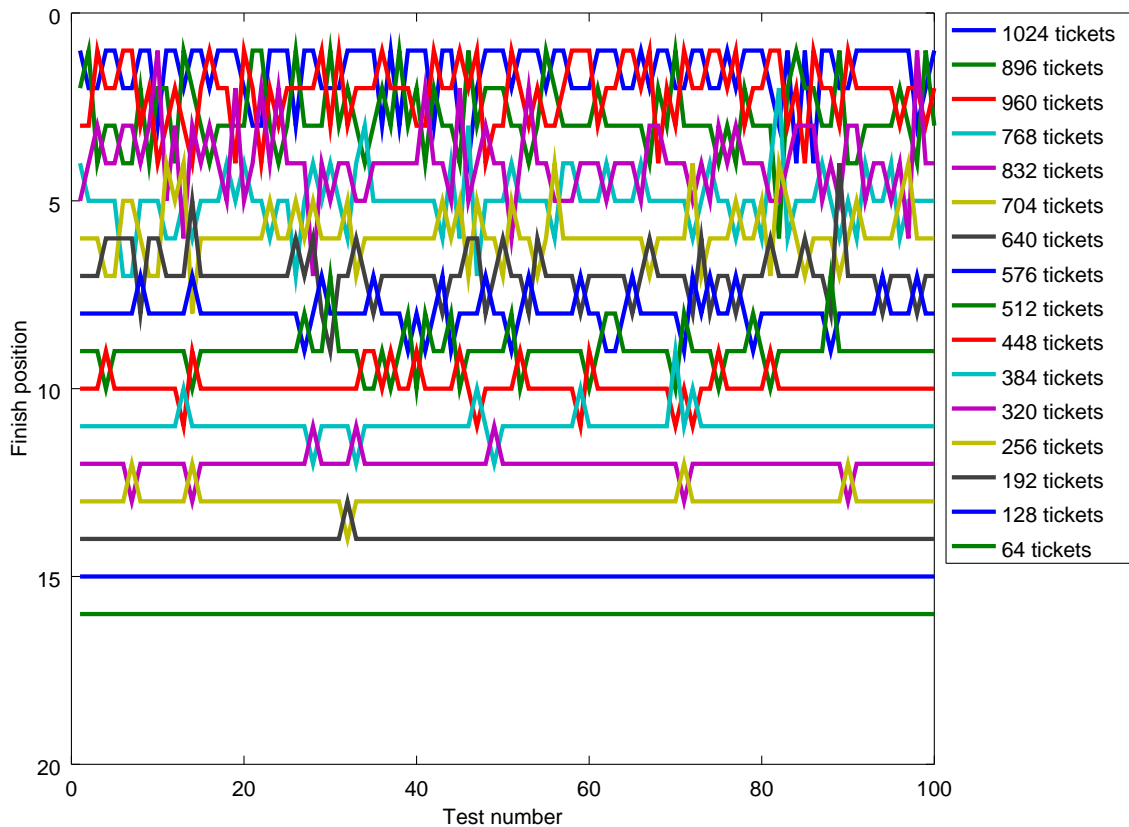
```

The Segment Tree works as follows. We initialize the `pass` for every process in the `pinit` function, before any process runs. Then the function `build` is called, just one time in the operating system lifetime, to initialize the whole tree. All of its leaves will each receive a `pid` from 1 to `NPROC`, in a traversal-ordered way (it means that if the leaves were a linked-list, following it, starting at the first leaf, would print all `pids` in ascending order), and every parent of the leaves will store the `pid` for the process that currently has the minimum `pass`. As every `pass` is set to infinity and the comparison is done with the \leq operator, the left operand will always win and we will end up having `pid 1` at the root.

4. Statistical analysis and conclusion

After all the modifications presented here, it is time to see if the scheduler is working as intended. To test it, the `lotterytests.c` file was created and configured as an `xv6` executable just like `usertests.c`. It works like the following: 16 processes are created with the job of counting from 0 up to 10^8 . The i^{th} process received $i \times NPROC$ tickets. This test was executed 100 times. A `cprintf` was purposely added to the `exit` function, in the `proc.c` file, to print the amount of tickets a process that just terminated had. This was later processed in the host operating system by an ordinary C program that produced output conforming to the GNU Octave syntax to generate the following graphic showing the order in which those processes terminated:

Please note that the *x-axis* represent the test number and the *y-axis* is the position between 1 and 16 that a process with a number of tickets (represented by the color) terminated. Also take into account that the processes were created sequentially from the one holding the least amount of tickets to the one holding the highest amount of tickets.



As a random algorithm, there is no guarantee in the order of execution of the processes. The only sure thing is that the higher the amount of tickets, the higher the probability of being chosen to run and the larger the amount of tests, the more this order will converge to the pattern established by that probability. Additionally, there is no way to trigger the processes all at once and this is also part of the reason that a process sometimes terminates before another one that has more tickets: it started running before. If the processes were created in the inverse order, from the one holding the highest amount of tickets to the least amount of tickets, this behavior would not happen so frequently.

Despite that, it is clear that the processes with the highest amount of tickets terminated first (again, with some variation among those with a similar quantity of tickets), and the ones with the lowest amount of tickets terminated last, as we expected.

References

- Cox, R., Kaashoek, F., and Morris, R. (2014). *xv6 - a simple, Unix-like teaching operating system*.
- Halim, S. and Halim, F. (2013). *Competitive Programming 3: The New Lower Bound of Programming Contests*.
- Waldspurger, C. A. and Weihl, W. E. (1994). Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA. USENIX Association.

Waldspurger, C. A. and Weihl. W., E. (1995). Stride scheduling: Deterministic proportional-share resource management. Technical report, Cambridge, MA, USA.