# Implementação de um escalonador de processos baseado em loteria

**Ivair Puerari**[1], **Jeferson Schein**[2]

[1]Universidade Federal Fronteira Sul(UFFS)
Chapecó – SC – Brasil

[2]Ciência da Computação
Universidade Federal Fronteira Sul (UFFS) – Chapecó, SC – Brasil

ivaair@hotmail.com, schein.jefer@gmail.com

***Abstract.*** *This article refers to the implementation of a lottery-based process scheduler, essentially important for an operating system, because it is its function to perform the distribution of resources so that everyone has the opportunity to use, given that we propose a solution to the problem, we describe implementation strategies and the step by step how it was carried out to develop it there have been changes in the structures and functions, especially in the development of the planning we had the need to implement a function of randomness. The operating system used for implementation and testing was the xv6 created by Massachusetts Institute of technology.*

***Resumo.*** *Este artigo refere-se a implementação de um escalonador de processos baseado em loteria,essencialmente importante para um sistema operacional, pois é de sua função realizar a distribuição de recursos de forma que todos tenham a oportunidade de usar, diante disso propomos uma solução para o problema, descrevemos as estratégias de implementação e o passo a passo de como foi procedido para desenvolvê-lo houve mudanças nas estruturas e funções, especialmente no desenvolvimento do escalonador se teve a necessidade de implementação de uma função de aleatoriedade. O sistema operacional utilizado para implementação e teste foi o xv6 criado pelo Instituto de tecnologia de Massachusetts.*

## 1. Introdução

Em um sistema operacional é de grande relevância se ter um escalonador de processos onde que por meio de um algoritmo o escalonador equilibra a utilização dos recursos de forma eficaz, decidindo quem deve ser o atendido. Uma tarefa ou até um usuário interativo, a partir da logica implementada em seu algoritmo, melhorando sua performance. Para desenvolvimento de um algoritmo eficaz é preciso saber o que esperamos do algoritmo, o que ele deveria fazer, e isso também depende em qual ambiente vai ser exposto, quando falamos em ambiente , falamos nas características que o sistema operacional tem, como área de aplicação e objetivo.

Ambientes podem ser divididos como sistema em lote, interativo e tempo real, dividido por suas particularidades e necessidades com essas informações é possível saber qual algoritmo é melhor de se implementar para a aplicação, como dito antes, cada algoritmo demonstra mais familiaridade em um ambiente distinto, e deve se levar em conta isso na hora da escolha.

## 2. Problema

O intuito deste artigo é relatar os acontecimentos da implementação de um escalonador de processo em ambiente interativo, levando como base o escalonamento por loteria, que por meio bilhetes de loteria, estes dado aos processos, cujo os prêmios são recursos do sistema. Onde é realizado um sorteio de um bilhete e o processo que conter o bilhete terá a oportunidade de usar o recurso. O sistema operacional escolhido para implementação e teste foi o xv6, sistema operacional criado pelo curso de Sistemas Operacionais no Instituto de Tecnologia de Massachusetts, nos Estados Unidos da América.

Como uma das politicas do escalonador de processos a ser desenvolvido, deve ser permitido passar a quantidade de bilhetes na instanciação dos processos, afim que processos que contenham os maiores números de bilhetes de loteria, tenham mais chances de serem sorteados.

## 3. Planejamento

O planejamento da solução para o problema, foi dividido em dois momentos, teórico e implementação.

Inicialmente foi feita a leitura do manual do xv6, para compreensão do funcionamento do sistema operacional, após , foi feito pesquisas para melhor entendimento do escalonamento por loteria, de como deve ser realizado a sua implementação, diante destes estudos, partimos para os códigos e estruturas do xv6, afim de entender o seu funcionamento, e bem como encontrar arquivos que eram importantes para a solução.

O planejamento de implementação, decorreu de forma que por escolha, achamos melhor nos concentrar em um primeiro momento na instanciação dos processos nas mudanças necessárias para o recebimento de uma informação externa enviada pelo usuário que seria quantidade de bilhetes do processo, nas structs e parâmetros de funções relacionadas. Após termos um processo com bilhetes , passamos a dar enfoque na implementação algoritmo baseado no escalonamento por loteria.

A peça chave para um bom escalonamento é de como é realizado o sorteio, para isso precisa de uma boa função de rand, onde deve ser escolhido um bilhete entre os dispostos, sabendo disso, foi o nosso próximo passo, desenvolver uma função rand que aleatoriamente nos entregasse um bilhete entre 1 e o máximo de bilhetes, assim com a implementação da função rand feita, conseguimos prosseguir com a implementação do algoritmo. Por fim realizamos testes probabilísticos, de modo que valida-se a solução.

## 4. implementação do escalonador

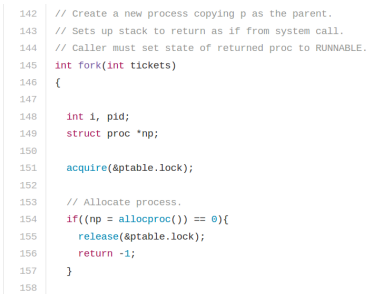
Os códigos do xv6 quem implicam no desenvolvimento, estão concentrados nos arquivos proc.c, proc.h. Que se distribui entre os outros arquivos, nas chamadas das funções como o de fork, bem como mudanças de estruturas e variáveis que dão suporte ao desenvolvimento. Nas próximas seções ira dar mais detalhes de como procedemos e as soluções propostas para a implementação do escalonador de processos.

### 4.1. Alterações no Fork

Primeiro foi necessário a atualização da função fork(), e o que vem a ser esse fork()? É a função que cria um processo novo, retornando a identidade (id) do pai dele, para que um

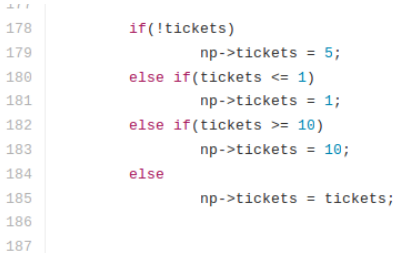
controle de paternidade entre os processos. A atualização no fork() foi algo simples, pois apenas foi adicionado um parâmetro de entrada da função, que é o número de bilhetes que esse processo irá ter quando ele estiver à espera de recursos para uso. Que o usuário pode atribuir uma quantidade de bilhetes ou por default foi definido 5 tickets por processo, assim como o número mínimo de tickets, e o número máximo de tickets ficou definido como 10, se acaso o usuário atribuir numero negativo ao processo, acreditamos que ele seja o ultimo a ter alguma prioridade e definimos 1 ticket para ele. Tendo assim 9 níveis de prioridade entre os processos, quanto mais tickets maior sua prioridade.

Prioridade é o quanto importante é um processo executar antes dos demais, mesmo que com o escalonador por loteria é por "sorte", pois é sorteado um ticket entre todos os existentes, mas conforme mais alto o nível de prioridade existe uma porcentagem maior desse processo ser escolhido para executar, por exemplo, existem 2 processos prontos para serem executados, o primeiro processo tem 1 ticket e o segundo tem 3 tickets, sendo assim, o primeiro processo tem 25% de chance de ser escolhido, assim como o segundo processo tem 75%. É uma divisão importante, pois existem processos que são mais importantes serem executados antes dos demais processos em sistemas operacionais.



```
142 // Create a new process copying p as the parent.
143 // Sets up stack to return as if from system call.
144 // Caller must set state of returned proc to RUNNABLE.
145 int fork(int tickets)
146 {
147
148   int i, pid;
149   struct proc *np;
150
151   acquire(&ptable.lock);
152
153   // Allocate process.
154   if((np = allocproc()) == 0){
155     release(&ptable.lock);
156     return -1;
157   }
158
```

**Figura 1. Paramêtro da função fork.**



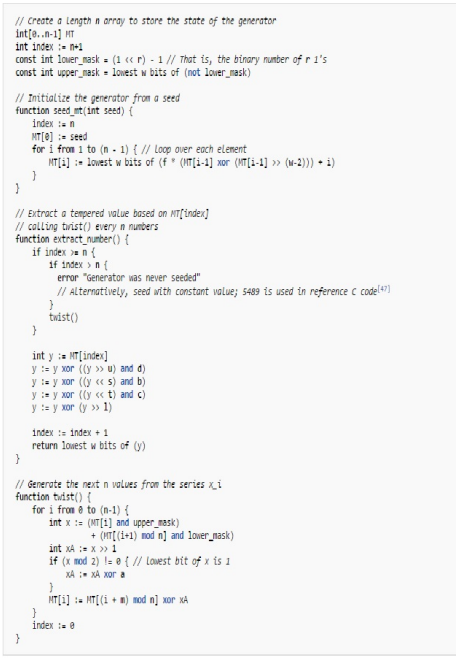
```
178         if(!tickets)
179                 np->tickets = 5;
180         else if(tickets <= 1)
181                 np->tickets = 1;
182         else if(tickets >= 10)
183                 np->tickets = 10;
184         else
185                 np->tickets = tickets;
186
187
```

**Figura 2. Definição de politica de bilhetes(tickets).**

## 4.2. Implementação da função Rand

Com o fork(), prioridade e número total de bilhetes atualizados e adaptados ao escalonador por loteria, é necessário uma função que faça o sorteio, e é claro que é necessário que cada bilhete deve ter a mesma porcentagem de ser sorteado, mas como fazer isso? Pois sabendo que não é algo que se possa resolver tão facilmente, então é utilizado o conceito de pseudo aleatoriedade que consiste em gerar número aleatórios a partir de uma base específica, mas sendo assim sempre que uma base ser utilizada os mesmos números aleatórios serão gerados, por exemplo, numa base 10, a função gera os valores (1, 75, 78,

40, 37), sempre que a base 10 for utilizada a função irá gerar a mesma sequência de valores (1, 75, 78, 40, 37), para contornar esse problema, para conseguir uma aleatoriedade maior, de tempos em tempos esse valor base é alterado para que assim gere uma sequência diferente de valores,entre o seed e o numero máximo de tickets que os processos prontos para executar tem, somados na função ticketCount. Sendo assim mais aleatório possível, pois aleatoriedade em computação é algo tão complicado de se gerar, que o próprio linux utiliza dados como interrupções e tempo que essas interrupções são geradas para assim conseguir valores aleatórios, mais aleatórios possíveis, mesmo não sendo aleatoriedade propriamente dita. E por esses motivos foi utilizado o algoritmo de Mersenne Twister.



```
// Create a length n array to store the state of the generator
int[0..n-1] MT
int index := n+1
const int lower_mask = (1 << r) - 1 // That is, the binary number of r 1's
const int upper_mask = lowest w bits of (not lower_mask)

// Initialize the generator from a seed
function seed_mt(int seed) {
    index := n
    MT[0] := seed
    for i from 1 to (n - 1) { // Loop over each element
        MT[i] := lowest w bits of (f * (MT[i-1] xor (MT[i-1] >> (w-2))) + i)
    }
}

// Extract a tempered value based on MT[index]
// calling twist() every n numbers
function extract_number() {
    if index >= n {
        if index > n {
          error "Generator was never seeded"
          // Alternatively, seed with constant value; 5489 is used in reference C code[47]
        }
        twist()
    }

    int y := MT[index]
    y := y xor ((y >> u) and d)
    y := y xor ((y << s) and b)
    y := y xor ((y << t) and c)
    y := y xor (y >> l)

    index := index + 1
    return lowest w bits of (y)
}

// Generate the next n values from the series x_i
function twist() {
    for i from 0 to (n-1) {
        int x := (MT[i] and upper_mask)
                + (MT[(i+1) mod n] and lower_mask)
        int xA := x >> 1
        if (x mod 2) != 0 { // lowest bit of x is 1
            xA := xA xor a
        }
        MT[i] := MT[(i + m) mod n] xor xA
    }
    index := 0
}
```

**Figura 3. Pseudocódigo do algoritmo de Mersenne Twister. Fonte: Wikipedia**

### 4.3. Alteração na função scheduler

Com todo o resto pronto e funcionando adequadamente se deu inicio as alterações no escalonador propriamente dito, primeiramente a criação de uma variável que a cada 500 iterações do escalonador, a seed(base) da função random é alterada (pelos motivos citados no parágrafo anterior), e em cada iteração é sorteado um bilhete para selecionar o processo. Como os processos estão em uma tabela de processos, é percorrido essa tabela para selecionar o processo que tiver o bilhete sorteado.

Para sortear um bilhete primeiramente é contado o número total de bilhetes, implementada em uma função chamada ticketCount, onde percorre toda a tabela de processo e conta o número de bilhetes dos processos “RUNNABLE”, e então o resultado do rand mod o número total de tickets da o número do bilhete sorteado, assim pegando o processo com esse ticket.

Cada processo tem um número de bilhetes, portanto o número do bilhete sorteado é considerado uma sequência, ou seja, se o processo 1 tem 8 tickets, ele terá os tickets 1 a 8, e o processo 2 tem 4 bilhetes, por sua vez terá os bilhetes 9 a 12. Então depois de ter

o número sorteado, percorre-se a tabela de processos para selecionar o processo que será executado no momento.

Quando estamos em um processo, é descontado do número sorteado o número de tickets do respectivo processo, e quando a variável que recebeu o valor sorteado, chegar a 0 ou menor que 0 estaremos no processo sorteado, assim selecionando o processo sorteado.

```
int ticketCount(){
        int count=0;
        struct proc *p;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->state == RUNNABLE){
                        count+=p->tickets;
                }
        }
        return count;
}
```

**Figura 4. Implementação função ticketCount**

```
void
scheduler(void)
{
  struct proc *p;
  int sum = 0, lot = 0, count = 0, seed = 1, maxTicket = 0;

  for(;;){
        // Enable interrupts on this processor.
        sti();
        count++;

        maxTicket = ticketCount();
        if (count >= 500){
                if (seed >= 2123456789)
                        seed = 1;
                if (count >= 200)
                        seed++;
        }else
                count++;

        Initialize(seed);

        lot = Extract() % maxTicket;

        // Loop over process table looking for process to run.y
        acquire(&ptable.lock);
        if(maxTicket > 0){
                for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                        sum += p->tickets;
                        if(p->state == RUNNABLE){
                                lot -= p->tickets;
                                if(lot < 0)
                                        break;
                        }
                }


                // Switch to chosen process.  It is the process's job
                // to release ptable.lock and then reacquire it
                // before jumping back to us.
                proc = p;
                switchuvm(p);
                p->state = RUNNING;
                swtch(&cpu->scheduler, p->context);
                switchkvm();
                // Process is done running for now.
                // It should have changed its p->state before coming back.
                proc = 0;

        }

        release(&ptable.lock);

  }
}
```

**Figura 5. Alterações feitas na função scheduler**

## 5. Testes aplicados no xv6

Como estratégia para testar se o escalonador de processo está atendendo a proposta, utilizamos um arquivo em c onde simulamos a criação de processos pela disputa da cpu, com números de bilhetes previamente já estipulados para cada um com um determinado tempo permitindo a concorrência entre os mesmos é feito, de modo que um tenha mais prioridade que o outro, para que probabilisticamente eles respondam conforme a quantidade de bilhetes, ou seja, que dependendo da quantidade de bilhetes que o processo tenha, ele tenha mais chances de ser o escolhido. Um exemplo seria se tivéssemos dois processos A e B concorrendo pelo tempo da cpu, e tivéssemos como máximo de bilhetes 0-99 , se darmos 0-74 bilhetes para o processo A e do 75 - 99 para o processo B, probabilisticamente ele deve tender em 75% do tempo ser o processo A em uso e 25 % processo B. EM um

algoritmo perfeito deveria nos dar como resposta esses números, aceitamos como correto, uma taxa de erro de 5% - 10% para mais ou menos na probabilidade pela quantidade de bilhetes que cada processo possui.

## 6. Conclusões

A implementação do escalonador de processos tem um impacto grande dentro do sistema operacional, afetando diretamente o desempenho para que seja um sistema operacional eficaz. Como a função de aleatoriedade é implementada influencia em seu desempenho, pois números devem sair conforme o os bilhetes retirados. Com o uso de aleatoriedade o escalonamento por loteria tende a ser uma probabilidade correta desejada, mas não há garantia, por fim quantos mais estes processos competirem pelo recurso sera mais provável que eles atinjam as porcentagens desejadas.

## 7. Refenrencias

Manual xv6 - xv6a simple, Unix-like teaching operating system. TANENBAUM, A. S. Sistemas Operacionais Modernos. 2. ed. São Paulo: Prentice-Hall do Brasil, 2003.