Lottery Scheduler no xv6: relatório de implementação e análise dos testes

Andrew M. Silva

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

andrewsaxx@gmail.com

Abstract. *The purpose of this paper is to document the implementation of a process scheduling algorithm called the Lottery Scheduler in the xv6 operating system. By understanding its operation, advantages and disadvantages, the entire development process is presented in detail and the tests result in data that are analyzed in a probabilistic way to obtain a metric of the efficiency of this algorithm.*

Resumo. *O objetivo deste trabalho é documentar a implementação de um algoritmo escalonador de processos chamado de Lottery Scheduler no sistema operacional xv6. Ao compreender seu funcionamento, vantagens e desvantagens, todo o processo de desenvolvimento é apresentado detalhadamente e os testes resultam em dados que são analisados de forma probabilística para obter uma métrica da eficiência deste algoritmo.*

1. Informação geral

Um escalonador de processos trata-se de um algoritmo aplicado em sistemas operacionais que possui como objetivo escolher quais processos do sistema serão executados ao considerar algum tipo de política de escolha. A partir disso, este trabalho objetiva explicar, analisar e discutir a implementação e testes de um escalonador de processos por loteria, também conhecido como Lottery Scheduler, de forma que fique claro o seu funcionamento e suas principais características, relacionando-as com o sistema operacional em questão, o xv6.

2. Metodologia e ferramentas

2.1. QEMU

Para que seja possível executar o xv6 várias vezes seguidas para a realização de testes, é utilizado um emulador e virtualizador de máquinas de código aberto, o QEMU. Para obtê-lo, basta acessar o link https://www.qemu.org/download e seguir os passos de acordo com o sistema operacional utilizado na máquina, que, no caso do autor, será o Ubuntu.

2.2. Instalando e configurando o xv6

O xv6 é um sistema operacional didático reescrito pelo Massachusetts Institute of Technology (MIT) com o propósito de utilizá-lo no ensino de tópicos relacionados a Sistemas Operacionais, tais como gerenciadores de memória e arquivos, escalonadores, etc, (Cox et al.). Considerando sua robustez e facilidade de compreensão, o xv6 foi escolhido para realização deste trabalho, o qual pode ser encontrado no link https://github.com/mit-pdos/xv6-public.

Primeiro é necessário instalar e atualizar todas as dependências necessárias para a execução do xv6, como descrito abaixo para o terminal do Ubunto 14.04:

```
sudo apt-get update
sudo apt-get install build-essential
sudo apt-get install gcc-multilib
sudo apt-get install qemu
sudo apt-get install git
```

Feito isso, basta clonar o repositório do link acima, acessar a pasta, limpar arquivos binários criados (caso existam) e executar o emulador QEMU:

```
git clone https://github.com/mit-pdos/xv6-public
cd xv6-public
make clean
make qemu
```

Ao fazer isso, será aberta uma janela com o emulador já executando o xv6, o qual será espelhado no terminal. Para encerrar o sistema, basta fechar a janela que foi aberta. No entanto, para agilizar o processo, também se pode utilizar o *make qemu-nox*, que, ao invés de abrir uma nova janela, apenas executa o emulador direto pelo terminal, exigindo apenas que, para encerrá-lo, sejam pressionadas as teclas Ctrl+A e depois a tecla X.

2.3. Lottery Scheduler

Desenvolver boas políticas para a escolha de processos para executar em um sistema operacional sempre foi um desafio. Dos muitos métodos atualmente existentes, este trabalho implementa o Lottery Scheduler clássico, que nada mais é que um algoritmo probabilístico que simula uma escolha por prioridade que é imune ao starvation, termo utilizado para caracterizar processos que estão em estado de inanição, ou seja, mesmo estando prontos para serem executados, nunca são executados por conta da política de escolha (Waldspurger 1994).

Seu funcionamento consiste em, ao criar um processo, definir uma quantidade de tickets a ele, considerando um valor mínimo e máximo, os quais podem ser definidos pelo desenvolvedor. Quando o escalonador é chamado, ele sorteia um valor aleatório de 0 até a soma de tickets dos processos prontos, percorre a tabela de processos e executa aquele que possui o ticket sorteado, considerando sempre o valor acumulado de tickets. Assim sendo, supondo uma tabela com 5 processos distintos prontos para executar no exemplo da Figura 1, cada um com uma quantidade diferente de tickets, o escalonador sorteia o valor 15 e escolhe o processo respectivo para execução, que, ao considerar o valor acumulado dos processos percorridos, descobre-se que o escolhido é o terceitro processo.

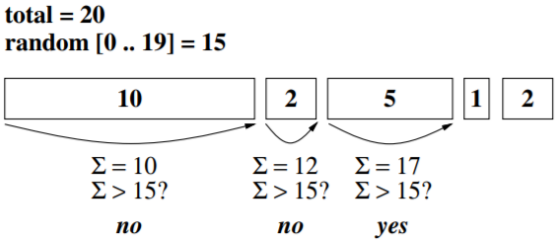

Figura 1. Exemplo de funcionamento do Lottery Scheduler

Portanto, é evidente que um processo com 1000 tickets possui mais chance de ser executado do que um processo com 10 tickets, por exemplo, o que evidencia a simulação da escolha por prioridade. Mas o que caracteriza este algoritmo como probabilístico é justamente o fato de que um processo com o número mínimo de tickets sempre terá uma probabilidade de ser escolhido, mesmo que pequena, pois ele nunca possuirá probabilidade nula. Essa é uma das maiores vantagens desse escalonador, pois essa característica o torna imune ao starvation, problema este que, em outros escalonadores, acaba por exigir a criação de políticas e algoritmos mais complexos para evitar isso.

Portanto, é fácil notar que, apesar de simples, o Lottery Scheduler é um algoritmo bem robusto e fácil de evoluir. Um exemplo de evolução é o estabelecimento de políticas para trocar tickets entre os processos, isto é, retirar tickets de processos pouco utilizados e enviar para processos mais utilizados e, dessa forma, modificar a prioridade dos processos em tempo de execução. Essa política é aplicada em algumas vertentes do Lottery Scheduler, sendo que existem muitas outras que utilizam diferentes tipos de soluções que funcionam melhor em determinados casos.

3. Implementação

Ao acessar a pasta do xv6, pode-se encontrar vários arquivos .c, .h e .asm, que são os arquivos que compõem o sistema. Cada arquivo possui implementada uma parte do sistema que vai ter uma função específica, como apenas apresentar a definição de funções do gerenciador de memória ou mesmo a implementação das funções pertinentes ao gerenciamento dos processos, e são esses os arquivos que interessam na implementação do escalonador.

3.1. Estrutura dos processos

O arquivo que mais será modificado é o proc.c, que é o que implementa as funções relacionadas ao gerenciamento de processos, mas ele não será o único a ser modificado. O arquivo que possui as definições dos objetos que serão interpretados como processos pelo sistema é o proc.h. Nele estão declarados os atributos que o processo possui, como os tickets. No entanto, o escalonador original do xv6 não implementa esse sistema de tickets e, portanto, os processos não possuem esse atributo. Logo, será necessário editar ele para incluir este atributo com o seguinte trecho de código:

```
...
struct proc {
  ...
  int tickets;                  // Tickets of process
```

```
  int schedCount;                // Times it was scheduled
};
...
```

O atributo *schedCount* é apenas um contador com o objetivo de armazenar quantas vezes o processo foi escolhido para ser executado. Dessa forma, será possível visualizar e comparar sua execução com outros processos para verificar se a escolha aleatória foi probabilística em relação aos tickets de cada processo. Assim sendo, este atributo é descartável e só está no código para a realização de testes e debugs.

Para continuar a implementação, é necessário indagar sobre alguns casos que podem ocorrer. É evidente que não deveria ser possível definir a um processo uma quantidade de 0 ou menos tickets, pois isso resultaria em uma probabilidade nula ou absurda, que não é algo característico do Lottery Scheduler. Além disso, seria incoveniente permitir a criação de um processo com 1 ticket e outro com 1.000.000.000 de tickets, pois é óbvio que as chances do primeiro processo ser executado seria absurdamente pequena.

Portanto, faz-se necessária a definição de uma quantidade mínima e máxima de tickets por processo para impedir a ocorrência desses casos. No entanto, qual os valores escolhidos vai depender da aplicação do sistema, nos quais um sistema em tempo real, por exemplo, talvez não precise de um grande intervalo de tickets visto que ele executará poucas aplicações. Não obstante, um sistema de uso geral, que é o caso que este trabalho aplica, é conveniente a definição de um intervalo um pouco maior, pois haverão vários processos concorrendo pelo processador. Os valores utilizados serão 10 e 50, mas nada impede de aumentar ou diminuir o intervalo. Assim sendo, ainda no arquivo *proc.h*, são definidas constantes que representarão os valores de máximo e mínimo, porém fora da struct do processo, como no trecho de código abaixo:

```
...
struct proc {
  ...
};

#define MIN_TICKETS 10
#define MAX_TICKETS 50
...
```

3.2. Criação de processos

Tendo já modificado a estrutura base dos processos, é possível seguir para o próximo passo: alterar a forma que os processos são criados. Isso é importante para que a quantidade de tickets do processo possa ser definido diretamente em sua criação, sem que seja necessário realizar chamadas para outras funções.

Nos sistemas UNIX a chamada de sistema utilizada para criar um processo é o *fork()*, que, na verdade, funciona como um duplicador de processos. A partir de um processo pai, ele pode executar este comando para criar um processo filho com código idêntico ao do pai. Após duplicar um processo, basta alterar o programa que ele executa através do comando *exec()* e, assim, ele será um processo idependente. Vale ressaltar que,

mesmo que os processos duplicados sejam semelhantes, o pid (id do processo) de ambos é totalmente diferente e nunca haverá dois pids iguais.

Continuando, já que o *fork()* é o comando que, de certa forma, cria os processos, assim como foi concuído anteriormente, é necessário alterá-lo para que, sempre que um processo for criado, seja definido uma quantidade de tickets para ele. O que se deseja é que, ao chamar essa função, seja passada como parâmetro a quantidade de tickets do processo. No arquivo defs.h estão definidos os escopos de várias funções e estruturas utilizadas no xv6 e nele a função *fork()* é declarada como *int fork(void)*, ou seja, sem nenhum argumento, então basta mudar da seguinte forma:

```
...
int fork(int);
...
```

Após isso, é necessário também mudar a forma que a chamada de sistema é criada no arquivo sysproc.c, que define que o *fork()* nunca recebe argumentos. Portanto, o novo códido da função *sys_fork()* deverá receber o argumento do tipo inteiro para mandá-lo representando os tickets do processo. O código resultante está abaixo:

```
...
int
sys_fork(void)
{
    int tickets;
    argint(0, &tickets);
    return fork(tickets);
}
...
```

Com isso, resta apenas modificar a implementação da função *fork()* para que ela receba os tickets e indique eles sempre que um processo for criado. Para tal, foi definido o parâmetro da função como *int tickets* e, em seguida, basta inicializar o atributo tickets do processo com o valor recebido. No entanto, uma regra deve ser respeitada: o número de tickets deve estar sempre entre os valores de mínimo e máximo definidos anteriormente e, se não for o caso, o valor deve ser modificado para se adequar ao intervalo, ou seja, se o valor for menor que o mínimo, usa-se o mínimo de tickets, se for maior que o máximo, usa-se o valor máximo de tickets. Isso vai previnir que um processo seja criado com um valor de tickets inesperado.

Além disso, o atributo schedCount deve ser inicializado em 0, mas não se esqueça que esse atributo apenas existe para que se possa ver facilmente os resultados do escalonador e, numa aplicação séria em que se pressupõe que escalonador já esteja implementado e testado, não há a necessidade desse atributo.

A partir de tudo isso, o código resultante será o seguinte:

```
...
int
fork(int tickets)
{
```

```
    ...
    np->schedCount = 0; // Initial value

    if(tickets < MIN_TICKETS)
        tickets = MIN_TICKETS;
    else if(tickets > MAX_TICKETS)
        tickets = MAX_TICKETS;
    np->tickets = tickets;
    ...
    cprintf("New process - Pid: %d Tickets: %d\n", np->pid,
    np->tickets);
    return pid;
}
...
```

Concluindo a etapa de alteração das estruturas e funções que envolvem a criação de processos, seria conveniente definir um valor default para a quantidade de tickets, isto é, tornar esse atributo opcional e definir um valor padrão caso ele não fosse passado, assim não seria necessário modificar todas as chamados fork do sistema, mas o puro C não oferece suporte para isso. Portanto, resta apenas acessar todos os arquivos que utilizem a função *fork()* para modificar suas chamadas definindo a quantidade de tickets. O valor aqui usado é zero, mas é garantido que esse valor estará no intervalo permitido, então pode ser definido qualquer valor (respeitando as características de valores do tipo *int*, é claro).

Também é necessário definir um número de tickets para o processo inicial, pois ele não é criado com um fork e, portanto, não possui tickets definidos. Para tal, é necessário alterar a função *userinit()*, que é a que cria o processo inicial, ainda no arquivo proc.c. Deve-se definir o número de tickets como o valor máximo e definir o schedCount como 0, assim, o escalonador terá o que necessita.

```
...
void
userinit(void)
{
  ...
  p->tickets = MAX_TICKETS; // Setting max of tickets
  p->schedCount = 0; // Initial value
  ...
}
...
```

3.3. Escalonador

Para iniciar essa etapa de desenvolvimento, é necessário fazer um levantamento do que precisará ser feito para que o escalonador funcione. Já se sabe que sempre que um ticket é sorteado, ele deve estar dentro do intervalo de zero até o total de tickets dos processos prontos para a execuçã, isto é, que possuem o estado *RUNNABLE*. Assim sendo, é evi-

dente que o somatório do valor total de tickets vai estar sempre variando, pois os estados dos processos vão mudar independentemente.

Logo, faz-se necessária a implementação de uma função que calcule o somatório de tickets apenas dos processos prontos para que ela possa ser chamada sempre que o escalonador for executado. Para tal, deve-se apenas acessar a tabela de processos, percorrê-la por completo e somar os tickets apenas dos processos com o estado *RUNNABLE*; esse somatório deve ser retornado no final da função. A implementação dela está definida no trecho de código a seguir:

```
...
int totalTickets(void){
    struct proc *p;
    int total = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == RUNNABLE)
            total += p->tickets;
    return total;
}
...
```

Feito isso, deve-se criar uma função para gerar valores randômicos, pois o xv6 não possui essas bibliotecas implementadas. Existem muitos métodos para a geração desses valores, mas na verdade não é possível gerar um número 100% aleatório, pois os valores que uma máquina pode criar são puramente determinísticos. Assim sendo, só é possível criar um algoritmo que gere valores pseudo-randômicos, isto é, números que parecem aleatórios mas não são, possuindo uma sequência que se repete sempre que é executada (Blum et al. 1986). No entanto, há uma forma de contornar esse problema definindo uma seed sempre que for gerar um valor, que é um valor que afeta a geração do proximo valor pseudo-aleatório.

Para tal, são implementadas duas funções em um arquivo separado, que aqui e nomeado de rand.c. A função *rand()* será responsável por gerar um valor e atualizar a seed, enquanto a função *srand()* deve dar ao usuário a possibilidade de alterar a seed manualmente. O método implementado é baseado no encontrado no site da Mirror[1], que está no código a seguir:

```
unsigned long int rand_next = 1;

int rand (){
    rand_next = rand_next * 1103515245 + 54321;
    return ((unsigned int)(rand_next / 65536) % 32768);
}

void srand(unsigned int seed){
    rand_next = seed;
}
```

[1] http://mirror.fsf.org/pmon2000/3.x/src/lib/libc/rand.c

Agora que é possível gerar bons valores pseudo-randômicos, pode-se seguir para o objetivo principal, a implementação do algortimo escalonador por loteria. Como o xv6 já possui um escalonador padrão, não é necessário criar um novo, basta apenas modificar o que é necessário.

A função *scheduler()* consiste em criar um loop infinito em que, dentro dele, a escolha de processo é feita. Sempre que o escalonador é chamado, deve-se obter o total de tickets prontos para serem executados e verificar se esse valor é maior que zero, pois, se não for o caso, significa que não há processos prontos. Se a condição for válida, deve-se sortear o valor de tickets no intervalo de zero até o total obtido, mas antes disso é importante alterar a seed com função *srand()* para melhorar a consistência da aleatoriedade.

Para tal, será passado como parâmetro a multiplicação do número de vezes que o escalonador foi executado, pelo número de ticks de clock que o sistema já executou. Como o número de ticks é um valor totalmente imprevisível, é evidente que a seed gerada é relativamente boa e, consequentemente, o valor randômico também será bom.

Agora a tabela de processos deve ser percorrida e, caso o processo esteja no estado *RUNNABLE*, uma variável que guarda o acumulado de tickets é somada com o valor do processo. Dessa forma, basta verificar se o valor acumulado é menor que o ticket sorteado e, caso negativo, este é o processo a ser executado.

Após escolhido o processo, deve-se incrementar seu atributo schedCount e, por fim, realizar a troca de contexto do processo, passo que já está implementado no escalonador padrão. Com isso, resta apenas parar a execução do escalonador com um *break* para impedir que os processos posteriores sejam executados, pois se isso não for feito, o escalonador executará todos os processos seguintes porque o valor acumulado sempre satisfará a condição definida anteriormente. Portanto, o código resultante está no trecho abaixo:

```
void scheduler(void){
    ...
    int chosen=1, total, times=0;

    for(;;times++){
        ...
        total = totalTickets(); // Get total of tickets
        if(total > 0){
            srand(ticks*times);
            chosen = rand() % total;
            int sum = 0; // Accumulated tickets
            for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                // Calculing Accumulated tickets
                if(p->state == RUNNABLE)
                    sum += p->tickets;
                if(p->state != RUNNABLE || sum < chosen)
                    continue;
                ...
                p->schedCount++;
                ...
```

```
                break;
            }
        }
        release(&ptable.lock);
    }
}
```

3.4. Visualizando os processos

Com isso, o escalonador por loteria está concluído e resta apenas a coletar e analisar os dados dos testes para verificar se ele funciona da forma esperada. Para que a coleta seja possível, o atributo countSched será (finalmente) usado; ainda no arquivo proc.c, a função *procdump()* pode ser usada dentro do xv6 ao pressionar as teclas Ctrl+P, ação que irá mostrar todos os processos do sistema e seus atributos. No entanto, para que seja possível ver os tickets e o schedCount dele, é necessário alterar a função, o que é feito apenas adicionando os atributos tickets e schedCount no *cprintf*, como no trecho que está logo abaixo:

```
void procdump(void){
    ...
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        ...
        cprintf("%d %s %s %d %d", p->pid, state, p->name,
        p->tickets, p->schedCount);
        ...
    }
}
```

3.5. Casos de testes

Para testar o escalonador, foi criado um programa chamado test.c, que cria N processos (definidos internamente) com valores aleatórios de tickets e os deixa executando em loop infinito. Para compilar o programa é necessário adicioná-lo à lista de arquivos fontes no arquivo Makefile, que é o arquivo que configura como os arquivos são compilados pelo comando *make* no terminal. O código completo do programa está abaixo:

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "rand.c"

#define N 10

int main(){
    int pids[N], i;

    for(i=0;i<N;++i) {
        int t = 10+(rand() % 40);
        if ((pids[i] = fork(t)) < 0)
```

```
            return 0;
        else if (pids[i] == 0)
            for(;;);
    }
    wait();
    return 0;
}
```

4. Resultados

Nesta seção são feitos os testes com o novo escalonador implementado, sendo necessário, obviamente, executar o xv6 pelo terminal e verificar se há erros no código. Para testar o escalonador basta executar o comando *test*, o qual executará o programa de teste implementado anteriormente. Ao fazer isso, será mostrado na tela dados sobre a criação dos processos, como definido na função *fork()*, contendo o identificador único do processo e a quantidade de tickets dele. Como o está definido no arquivo test.c que ele criaria 10 filhos, são criados 11 processos no total (fora o init e o sh), os quais podem ser visualizados na imagem abaixo:

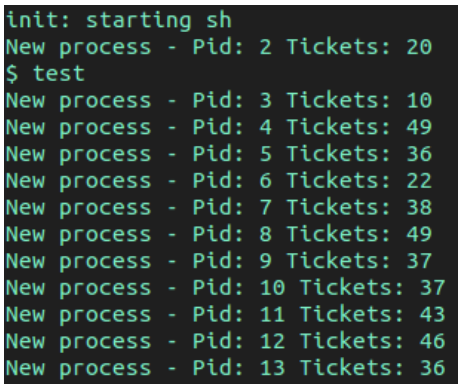

Figura 2. Criando os processos

A partir disso, já é possível calcular qual a probabilidade de escolha de cada um dos processos considerando que, como há apenas os processos filhos do test sendo executados, apenas eles serão utilizados nos cálculos. Como já se sabe que o total de tickets distribuído é 393, pode-se usar este valor para calcular a probalidade de execução de cada um dos processos. Organizando os dados em uma tabela de distribuição normal, obtêm-se o seguinte:

PID	Tickets	Probabilidade
4	49	12,48%
5	36	9,16%
6	22	5,59%
7	38	9,67%
8	49	12,48%
9	37	9,41%
10	37	9,41%
11	43	10,94%
12	46	11,70%
13	36	9,16%
Total	393	100%

Tabela 1. Tabela de distribuição dos processos criados

A tabela apresenta a probabilidade de execução de cada processo e uma das formas de saber se o escalonador está funcionando corretamente é deixar o programa executar por alguns minutos, coletar seus dados e fazer uma distribuição normal com essa amostra. Ao invés de calcular uma probabilidade, será calculada uma taxa de excecução em porcentagem para cada processo e, se as taxas forem próximas às probabilidades encontradas anteriormente, ficará evidente o funcionamento correto do algoritmo. Ao pressionar Ctrl+P, é possível coletar o atributo schedCount dos processos, o que pode ser visto na figura abaixo:

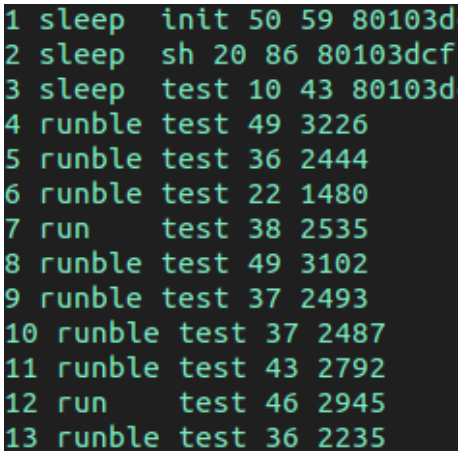

Figura 3. Dados sobre os processos criados

Ao observar o resultado, é possível perceber que processos com a mesma quantidade de tickets foram executados um número de vezes diferentes, fato que já é esperado já que a escolha do processo é aleatória. No entanto, ao comparar um desses processos com um outro que possua número de tickets diferente, é evidente que a diferença de execução entre eles aumenta. Isso é mostra que, de acordo com o número de tickets, um processo pode ser executado mais ou menos vezes, que é exatamente o que se buscava com esse algoritmo. Se esses dados forem organizado em uma tabela de distruibuição normal e o cálculo da taxa de execução for feito e comparado com a probabilidade determinada anteriormente, o resultado será o seguinte:

PID	schedCount	Taxa de execução	Probabilidade	Diferença
4	3226	12,72%	12,48%	0,24%
5	2444	9,49%	9,16%	0,33%
6	1480	5,57%	5,59%	0,02%
7	2535	9,85%	9,67%	0,18%
8	3102	12,05%	12,48%	0,43%
9	2493	9,69%	9,41%	0,28%
10	2487	9,66%	9,41%	0,25%
11	2792	10,85%	10,94%	0,09%
12	2945	11,44%	11,70%	0,26%
13	2235	8,68%	9,16%	0,48%
Total	25739	100%	100%	2,32%

Tabela 2. Tabela de distribuição dos processos executados

É fácil ver que a taxa de execução dos processos é muito próxima à probabilidade de execução dos mesmos, sendo que, considerando que a diferença total entre a taxa de execução e a probabilidade determinística seja 2,32%, obtemos uma variância de 0,23% (dividindo pelo número de processos, que é 10), mostrando que o algoritmo está funcionando probabilisticamente correto.

5. Considerações finais

Com os resultados positivos dos testes, fica claro que o escalonador foi implementado de forma coerente às regras abstratas do Lottery Scheduler. A taxa de execução dos processos revela que a quantidade de tickets realmente afeta na escolha do escalonador e que é bem simples definir diferentes prioridades para os processos sem correr o risco de starvation, o que é uma grande vantagem.

Além disso, existem outras vertentes do Lottery Scheduler que otimizam essa questão de prioridade, mas também há outras formas de realizar a escolha de processos sem necessitar de um número aleatório. A robutez desse algoritmo é o que permite que tantas otimizações e vertentes sejam feitas, provando que ele é aplicável em muitos tipos de sistema operacionais. Claro, existem outros algoritmos que são melhores que este em alguns aspectos, mas a escolha de qual algoritmo utilizar vai depender muito da aplicação usada para a máquina e/ou sistema. Dessa forma, pode-se concluir que todos os escalonadores são válidos, mas cada um possui suas particularides, vantagens e desvantagens em diferentes tipos de aplicações, cabendo ao projetista o papel de escolher o método que mais se encaixa em suas necessidades e especificações.

Referências

[Blum et al. 1986] Blum, L., Blum, M., and Shub, M. (1986). A simple unpredictable pseudo-random number generator. *SIAM Journal on computing*, 15(2):364–383.

[Cox et al.] Cox, R., Kaashoek, M., and Morris, R. Xv6, a simple unix-like teaching operating system. pdos. csail.

[Waldspurger 1994] Waldspurger, Carl A. e 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.