Implementação de escalonadores de processos baseados em Loteria e Passos largos

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 two schedulers for processes based on lottery and stride, essentially important for an operating system, because it is his function to do the distribution of resources so that everyone have the opportunity to use. Therefore we propose a solution to the problem, we describe implementation strategies and the step by step how it was carried out to develop them, the changes in the structures, functions, and the specifics of each type of schedulers. 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 dois escalonadores de processos baseados em loteria e passos largos, 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-los, as mudanças nas estruturas, funções e as especificidades de cada tipo de escalonamento. 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 haver um escalonador de processos aonde 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 , refere-se as 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 possıvel 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.

Sera visto a seguir dois tipos de escalonamentos de processos, um baseado em loteria e outro em passos largos. A problematização, e os critérios contidos em cada tipo de escalonamento, bem como a sua implementação e analise de resultados.

2. Problema

O intuito deste artigo é relatar a forma de planejamento e implementação de dois escalonadores de processos baseado em passos largos e loteria, como trabalhos propostos na disciplina de Sistemas Operacionais. O ambiente de desenvolvimento utilizado como sistema operacional tipo Unix XV6, criado pelo instituto de Tecnologia de Massachusetts, nos Estados Unidos da América.

Em um primeiro momento foi desenvolvido o escalonamento de processos baseado em loteria, que por meio bilhetes de loteria, estes dados aos processos, cujo os prêmios são recursos do sistema. Assim, é realizado um sorteio de um bilhete e o processo que conter o bilhete terá a oportunidade de usar o recurso.

Como politica do escalonador de processos baseado em loteria, 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. Após o desenvolvimento no ambiente XV6 ser compatível e rodável, ou seja, entregar um bom resultado, quando utilizado escalonador de processos baseados em loteria. O próximo passo é o desenvolvimento do escalonador de processos baseados em passos largos.

O escalonamento por passos largos, assume o papel de um escalonamento, que deterministicamente aloca espaço de tempo proporcional, para execução da tarefa proposta do processo, variando conforme a quantidade de bilhetes que o processo possui. Assim cada processo deve conter uma quantidade fixa de bilhetes na instanciação do processo, assim calcula-se o passo (stride) de cada processo. Logo é necessário adicionar a sua passada, o valor anteriormente contido mais a sua passada, passada essa encontrada sendo o quociente entre a divisão de um numero constante pelo seu numero de bilhetes contidos.

Inicialmente cada processo tem passada igual a zero, ”stride.pass = 0”é um exemplo, com isso o escalonador escolhe o processo com a passada menor, para que seja o processo executado, portanto, qualquer processo entre todos os processos podem inicialmente ser o escolhido. De acordo a abordagem implementada, a prioridade diz qual o processo a ser o primeiro escolhido, a partir disso, é feito a operação de soma da passada do processo e assim sucessivamente.

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.

Contudo, Para uma solução ao problema de implementação de um escalonador de processos, primeiramente, foi realizado um estudo sobre o método de escalonamento passos largos, quais suas especificações e requisitos para o desenvolvimento.

Como anteriormente, em um primeiro momento, foi realizado um estudo em cima do manual do XV6, para que fosse redigido o artigo ”Implementação de um escalonador de processos baseado em loteria”. Foi possível que pulássemos esta etapa, e que houvesse uma maior concentração na implementação. Após buscas, com o objetivo de obter o entendimento do método de escalonamento por passos largos, foi verificado quais blocos de códigos deveria ser mudado ou deixados como eram, que não implicavam na solução.

A proposta para planejamento foi seguir as características do escalonamentos em passos largos. Logo, iniciando os processos com uma quantidade de bilhetes, onde o usuário ou qualquer outro criador de processo, passe como informação a quantidade bilhetes que aquele processo ira conter.

Após isso, a construção de um algoritmo ou mecanismo para que se calcule o passo de cada processo, bem como, inicie os mesmos. Em outro momento, haverá a escolha entre como proceder na escolha de qual processo ser escolhido quando todos tem sua passada iguais, e posteriormente realizar a atribuição da sua nova passada, que seria a soma de sua passada mais o passo, originado pelo numero de bilhetes atribuído ao processo.

4. implementação do escalonador de processos baseado em loteria

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).

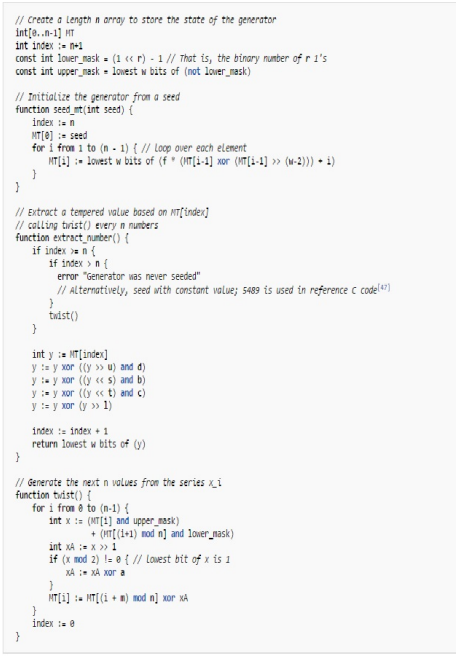
4.2. Implementação da função Rand

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.

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.

Torna-se 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. implementação do escalonador de processos baseado em passos largos

Na implementação, foi constatado que os blocos de códigos que seriam necessários mudar para que a solução fosse eficiente, se encontram nos arquivos ”proc.c”e ”proc.h”. Estes arquivos também utilizam outros arquivos que dão suportes as funções contidas.

Como anteriormente proposto, foi inciado a implementação para que os processos sejam possíveis conter os bilhetes, bem como, atribuir os o bilhetes ao processo. Nas próximas seções deste artigo sera relatado de como houve essas mudanças, informando suas particularidades, e disponibilizando os trechos de códigos, para que ilustre melhor a explicação de como procedeu a implementação do escalonador de processos baseado em passos largos.

5.1. Bilhetes para processos

Houve a necessidade de mudanças na função fork no arquivo ”proc.c”. A função fork é a responsável pela criação de processos, e tem como seu retorno o id do processo pai, criador do novo processo, o processo que originou o chamado da função.

Para a solução funcionar, não foi necessário criar um parâmetro dentro da estrutura processo. Mas, repassar a informação quando realizado uma chamada para a função fork, como parâmetro da função, um numero inteiro, representando os bilhetes do novo processo.

Cada processo pode conter no máximo 1000 bilhetes,e no mínimo 50 bilhetes. Caso houver algum erro, de não haver o numero de bilhetes de um processo, como default, ira ser utilizado um numero médio de bilhetes, a quantidade de 500 bilhetes para o processo.

Dentro da função fork existe uma chamada para outra função, a função allocproc , nesta função realmente é criado um novo processo, onde tem os parâmetros do processo instanciado. Sabendo disto, Foi passado a quantidade de bilhetes como parâmetro para a função allocproc, para que a quantidade de bilhete que foi dado ao processo, esteja no momento de criação do processo, já que os bilhetes significam o quanto um processo é prioritário ou não. Prioridade seria ter o privilegio de obter o direito sobre o recurso requerido para que o processo termine suas tarefas rapidamente.

```
int
fork(int tickets)
{
  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc(tickets)) == 0)
    return -1;

  // Copy process state from p.
  if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);
  np->cwd = idup(proc->cwd);

  pid = np->pid;
  np->state = RUNNABLE;
  safestrcpy(np->name, proc->name, sizeof(proc->name));
  return pid;
}
```

Figura 6. Paramêtro da função fork.

5.2. Passo e passada de um processo

Passo em um processo, é o quociente de uma divisão entre um numero constante e a quantidade de bilhetes que o processo possui. Neste caso, foi utilizado um numero constante 50000, escolhido arbitrariamente. O passo é o resultado da operação de divisão, do quociente da divisão.

Para exemplificar, antes de demonstrar a implementação da solução, considere que exista três processos, A, B, C. O processo A com 100 bilhetes, processo B com 250 bilhetes e processo C com 50 bilhetes. O processo A ira ter um passo de 500, processo B um passo de 200 e o processo C com passo 1000. Como dito anteriormente um escalonamento em passos largos, os processos tem sua passada todos iniciados com zero, para uma forma didática, iremos elaborar a ordem de forma que foi exposto os processos, em ordem léxica. Assim, o processo A será o primeiro, com sua passada que é igual a 0, torna-se 0 + passo, ou seja, passo igual a 500, logo, é a vez de ir para o menor, como o escalonamento

por passos largos pede. Com isso e por ordem léxica, o processo B atualizara-se com sua passada que antes era zero para 200, após isto, é escolhido o processo C que tem sua passada mantida igual a 0 ainda, que é atualizada para 1000.

É possível ver, que em uma próxima rodada, o processo B com a menor passada no momento, seria o escolhido, e no mínimo por duas rodadas sendo o escolhido. Então, quanto maior o numero de bilhetes, menor é sua passada, ou seja, mais tempo utilizando do recurso para a tarefa proposta pelo processo.

5.2.1. Passo

Na implementação,para que um processo possa conter um passo, foi necessário que o fosse atribuído um novo parâmetro dentro da estrutura processo localizado no arquivo ”proc.h’. No arquivo ”proc.h”foi criado dentro da estrutura a variável step.

Como cada processo tem seu próprio passo, partindo da quantidade de bilhetes pertencentes. Intuitivamente foi decidido que dentro da função allocproc, já que instancia os parâmetros da estrutura processo, para que seja feito a atribuição do valor de passo do processo. O atributo bilhetes é passado como parâmetro da função, logo, a partir da utilização de algumas condições de fluxos, foi implementada a operação para descobrir o passo do processo e atribuído ao step o seu valor de passo.

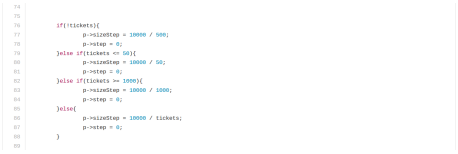


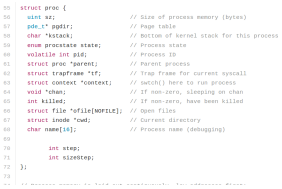
Figura 7. Função allocproc, operação implementada para o passo.

5.2.2. Passada

Para implementação da passada, foi feito um estudo onde melhor se encaixaria as mudanças. Foi verificado que propriamente na função scheduler é realizado a escolha do processo na tabela de processos do sistema operacional. Mas, anteriormente foi realizado a atribuição de uma nova variável na estrutura processo, chamada sizeStep, representando a passada, modificada no arquivo ”proc.h”.

Após isso, e como especifidade do problema, toda passada deve ser igual a 0 para todos os processos. Assim, logo quando é atribuído o valor de passo do processo, também é atribuído a sua passada o valor 0 para a variável sizeStep que pode ser verificada na função scheduler **Figura 2**.

Deve se levar em conta que todo processo inicialmente terá 0 como sua passada, logo, deve haver um critério de desempate. Para que não haja influencia ou que seja o mínimo possível, foi decidido que a primeira passada 0, de qualquer processo seria o escolhido, ou seja, o primeiro processo encontrado com passada 0 é o escolhido.



```
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
  volatile 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 step;
        int sizeStep;
};
```

Figura 8. Varivel step (Passo) e sizeStep(Passada) na estrutura processo.

A implementação para esta logica ocorreu de forma simples, comparamos todos os processos com a utilização de condições de fluxos, dentro de um for que passa pela tabela de processos do sistema operacional. Com o processo escolhido, é necessário ser atualizado a passada do processo,então foi implementado algo como se fosse uma flag, para que quando fosse encontrado o processo com a menor passada, executa-se este bloco de código para que seja feita a operação.

Assim, a passada do processo é somado com o passo do processo. Na possibilidade de haver um overflow de inteiros, pela causa de um acúmulo grande de somas, por ser maior do que a quantidade de memoria que um inteiro tem, tratamos de forma simples, por ser algo difícil de acontecer, mas possível, para que caso acontece o processo possa ser reiniciado a contagem de sua passada.

```
      int min;
      struct proc *p;
      struct proc *minP;
      for(;;){
          // Enable interrupts on this processor.
              sti();

              acquire(&ptable.lock);
              minP = 0;
              min = 5000000; // processo com uma passada maior que 5000000 nao sera mais executada

              for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                      if(p->state == RUNNABLE && p->step < min){
                              min = p->step;
                              minP = p;
                      }
              }

              // Switch to chosen process.  It is the process's job
              // to release ptable.lock and then reacquire it
              // before jumping back to us.

              if(minP){
                      proc = minP;
                      proc->step += proc->sizeStep;

                      proc->state = RUNNING;
                      switchuvm(proc);

                      swtch(&cpu->scheduler, proc->context);
                      switchkvm();

                      // Process is done running for now.
                      // It should have changed its p->state before coming back.
                      proc = 0;
              }
              release(&ptable.lock);
```

Figura 9. Função scheduler, operação de soma de passada.

6. Testes aplicados no xv6 e Analise de resultados

Os testes realizados aconteceram em dois momentos, primeiro teoricamente um teste de mesa, para encaminhar as implementações, bem como, dar garantia que se tem uma logica. Após a implementação, foi criado um arquivo ”test.c”. No ”test.c”, foi criado com o intuito de simular a interação de um usuário com o sistema operacional. No arquivo foi criado diversos processos com diferentes quantidades de bilhetes, imprimindo sua identificação e quantidade de bilhetes, afim de poder medir se conforme a quantidade de bilhetes do processo, ele terminaria de forma mais rápida , devida a prioridade dada pelo numero de bilhetes que contém.

6.1. Testes aplicados no xv6 com escalonamento de processos baseado em Passos Largos e Loteria

Houve a realização de um teste, em um primeiro momento, com o sistema operacional XV6 original, sem mudanças. Logo após isto, foi realizado um teste , fazendo o uso do

mesmo protótipo de teste, no sistema operacional XV6 que foi feita a implementação do escalonador de processos baseados em passo largos.

Ocorreu o uso de um timer externo, como ferramento para medição de quanto tempo cada sistema operacional levou para executar o teste. Visto isso, foi possível notar que houve uma diferença de 0,5 segundo, para o termino do segundo sistema operacional aquele que tinha mudanças no seu escalonador para o primeiro sistema operacional XV6 original.

Para o escalonamento de processos baseado em loteria , foi seguido o mesmo planejamento, em um primeiro momento, foi aplicado o mesmo protótipo de teste utilizado no escalonamento por passos largos, em um sistema operacional sem mudanças. Após, foi aplicado no sistema operacional que possui a implementação do escalonador de processos baseado em loteria.

Com a utilização da ferramenta do timer externo, houve uma diferença de 0,8 segundos entre os dois momentos. Assim, resultando que o sistema operacional com a implementação do escalonador de processos baseado em loteria, sendo o primeiro a terminar a execução.

6.2. Analise

Com os testes aplicados individualmente, para cada um dos dois escalonadores propostos em aula para o desenvolvimento, e contudo verificando as mudanças que foram dadas ao sistema operacional XV6 com as aplicações dos escalonadores. É possível afirmar, que enquanto analise de qualquer uma das duas aplicações implementadas no sistema operacional XV6 para o XV6 sem mudanças, traz resultados melhores em relação a desempenho e performance.

Quando se fala em qual aplicação tem o melhor desempenho, a aplicação do escalonador de processos baseado em passos largos se destaca. Logo, por possuir uma diferença de apenas 0,5 para o XV6 sem mudanças, enquanto, a aplicação de escalonamento de processos baseado em loteria, tem uma diferença de 0,8 segundo. Assim, é possível afirmar que a utilização do escalonador de processos baseados em passos largos gera um ganho de performance.

7. 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. O escalonador influencia diretamente no tempo de resposta dos processos, e quanto mais a quantidade de processos crescem, mais performance é requirida.

O escalonamento de processos baseados em passos largos, traz de uma forma determinística a essa alocação de recursos. Em testes foi verificado que houve um ganho em performance quando houve a utilização do escalonamento de processos baseados em passos largos. Demonstrando que a utilização de uma estratégia de escalonamento, deve ser escolhida de forma sensata, para que o sistema operacional seja eficiente em suas tarefas.

8. 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.
Waldspurger, Carl A.; Weihl William E.; Stride Scheduling:Deterministic Proportional-Share Resource Management. June 22, 1995.