Implementação de Escalonador por Passos Largos no Sistema Operacional Didático Xv6

Felipe Chabatura Neto[1]

[1]Departamento de Ciência da Computação - Universidade Federal da Fronteira Sul (UFFS)
Chapecó – SC – Brasil

felipechabat@gmail.com

Abstract. *This article describes the planning and implementation of the Stride Scheduling algorithm in the didactic operational system Xv6, reporting the process since the operational system analysis till the new scheduler test phase.*

Resumo. *Este artigo descreve o planejamento e implementação do escalonador de processos por Passos Largos (Stride scheduling) no sistema operacional didático Xv6, relatando o processo desde a análise do sistema operacional até a fase de testes do novo escalonador.*

1. Introdução

O escalonador é o componente do sistema operacional responsável por decidir quais processos são os próximos a utilizarem o processador para serem executados. Este trabalho consiste na implementação de um escalonador por loteria no sistema operacional didático Xv6.

1.1. O Xv6

O Xv6 é um sistema operacional didático desenvolvido em 2006 no curso de sistemas operacionais do MIT (*Massachusetts Institute of Technology*). O sistema é baseado no Unix Versão 6 e foi escrito na linguagem C ANSI [1].

1.2. Escalonamento por Passos Largos

Escalonamento por passos largos é um algoritmo determinístico para administrar disputas por recursos, no caso de um escalonador de processos, tempo de processamento. Cada cliente (entidade que disputa por determinado recurso) recebe inicialmente uma quantidade de bilhetes, que determinam a prioridade que o cliente tem sobre seus concorrentes na disputa pelo recurso. Para determinar a quantidade de tempo em que cada um vai utilizar o recurso, é calculado um atributo chamado de passada. A passada é uma unidade de tempo virtual, e é determinada pela divisão de uma constante (eg. 10000) pela quantidade de tickets que o cliente possui. Logo, quanto maior o número de tickets, menor a passada. A cada vez que um cliente utiliza do recurso concorrido, uma nova variável é atualizada, o passo. O passo é incrementado com o valor da passada do cliente a cada vez que ele utiliza do recurso. Como o próximo cliente a utilizar o recurso é sempre o com menor passo, os clientes com maior número de tickets e consequentemente menor passada, utilizam do recurso por maior quantidade de tempo, de maneira determinística. [2]

2. Planejamento e implementação

A primeira etapa se deu pela análise do sistema operacional Xv6, mais especificamente de seu escalonador, e do planejamento dos componentes que teriam que ser alterados para a implementação do novo escalonador.

O escalonador original do xv6 possui um funcionamento muito simples: Ele executa uma varredura linear pela tabela de processos, procurando o primeiro processo disponível para ser executado, o escalona para que ele inicie e reassume quando o processo transfere o controle de volta para o escalonador, que busca o próximo processo a ser executado.

A primeira alteração realizada no código do xv6 foi a alteração da estrutura dos processos, para que estes possuam os novos atributos necessários para a implementação do algoritmo: A quantidade de tickets, a passada, e o passo (propriedades *tickets*, *stride* e *pass* na estrutura *struct proc* no arquivo *proc.h*). Além disso, foram definidos uma constante para o cálculo da passada de cada processo e uma quantidade padrão de tickets (constantes *CONST* e *DEFAULT* no arquivo *proc.h*).

Para que o escalonador remova o processo com menor passo a cada vez em que for ativado, é necessário que haja uma estrutura de dados capaz de prover este tipo de operação. Para esta implementação, utilizou-se de uma *heap* binária, capaz de realizar novas inserções e deleções em tempo logarítmico, além de acesso ao elemento com menor passo em tempo constante (*O(1)*). Criou-se uma biblioteca com funções de extração e inserção para a heap binária, esta que foi incluida no arquivo *proc.c*, onde fica o código do escalonador, e adicionada ao *MAKEFILE* para compilar apropriedamente com o resto do código do xv6. A heap binária foi implementada utilizando um vetor de referências a processos, e utilizando da propriedade *pass* de cada processo para manter suas propriedades, de forma a manter sempre o elemento com menor passo sempre como raiz. Em caso de empate nos valores de passo, o primeiro elemento a ser inserido é o que vai ser escalonado.[3]

Em seguida, de maneira a atribuir valores diferentes de tickets para cada um dos processos criados, a função de fork original do xv6 foi alterada, de modo a passar como parâmetro o número de tickets que o processo que está sendo criado irá receber. Isto foi realizado alterando a função fork, a chamada de sistema fork e suas respectivas declarações (função fork no arquivo *proc.c*, chamada de sistema *sys-fork* no arquivo *sysproc.c*, declaração da função fork no arquivo *defs.h* e declaração da chamada de sistema no arquivo *user.h*). Além disso, todas as chamadas de fork foram alteradas, de modo a passar o valor 0, que foi definido como indicador de prioridade padrão para processos do sistema.

O próximo passo, foi fazer com que os processos que estão sendo criados, tenham seus atributos apropriadamente inicializados e sejam inseridos na heap. Para isso, dois fragmentos de códigos foram alterados, o primeiro na função *userinit* no arquivo *proc.c* onde o primeiro processo do sistema é criado, lá o valor padrão de tickets é atribuido, o passo é zerado, a passada é calculada, e o processo é inserido na heap. O segundo, no mesmo arquivo, é na função fork, por onde os demais processos passam a serem criados, lá, a quantidade de tickets é atribuída de acordo com o valor passado como parâmetro, se 0, o valor padrão é atribuído, se não, é atribuído o própio valor recebido. Também é

zerado o passo e calculado a passada, além do processo ser inserido na heap.

Em seguida, foram tratados os momentos de execução do sistema onde um processo deve ser inserido na heap, são eles: Quando um processo que está executando sai da cpu e volta a ficar RUNNABLE (função *yield* no arquivo *proc.c*) e quando um processo que está dormindo é acordado e volta a ficar RUNNABLE (função *wakeup1* no arquivo proc.c). Também foi necessário tratar os momentos em que um processo deve ser removido da heap (Além de quando ele é escalonado), sendo estes: quando um processo a ser morto pela função *kill* está no estado RUNNABLE (função *kill* no arquivo *proc.c*) e quando um processo é colocado para dormir estando no estado RUNNABLE (função *sleep* no arquivo *proc.c*).

Depois disso, foi alterado o funcionamento do escalonador em si (função *scheduler* no arquivo *proc.c*). Seu novo funcionamento se dá da seguinte maneira: Se a heap não estiver vazia, extrai o elemento com menor passo. Calcula seu novo passo, incrementando o valor da passada. Executa o processo.

```
acquire(&ptable.lock);
if(last > 0){ //Caso a heap não esteja vazia
  p = extract(1, heap, last--); //Extrai o de menor passo
  if(p->state == RUNNABLE){
    p->pass = p->pass + p->stride;  //Acumula a passada no passo
    p->scheduled++; //Conta como escalonad (Para teste)

    // 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 1. Trecho do código do escalonador responsável pela extração do processo a ser escalonado e atualização de seus atributos.

3. Testes

Para testar o novo escalonador, um simples programa foi desenvolvido e adicionado ao xv6 (programa *stridetest*, adicionado ao arquivo *MAKE* para ser incluido no xv6). O programa de testes tem um funcionamento muito básico: a partir dele, um determinado

número de processos filhos são criados, com diferentes números de tickets. Esses processos básicamente percorrem um loop gigantesco sem fazer nada, com o único intuito de gastar tempo enquanto o escalonador é analisado.

Para poder se ter uma métrica e avaliar o desempenho do novo escalonador, uma nova propriedade foi adicionada na estrutura dos processos, a propriedade *scheduled*, que indica quantas vezes determinado processo foi escalonado. Esta propriedade é atualizada a cada vez que o processo é escalonado. Para poder se observar as propriedades dos processos durante sua execução, uma alteração na função *procdump* (no arquivo *proc.c*, esta, é ativada com as teclas *ctrl + p* do teclado e exibe informações sobre os processos que estão ativos no momento) foi realizada, para que esta mostre os valores de passo, passada , tickets e a quantia de vezes que o processo foi escalonado. Observando os valores de passo dos processos criados pelo programa de teste enquanto estes executavam, foi possível perceber que os processos com maior número de tickets, e consequentemente menor passada, eram escalonados proporcionalmente mais vezes do que os outros, indicando o correto funcionamento do escalonador.

```
for(i = 0, num = 50; i < NPROC_T; i++, num += 50){
  printf(1, "Process %d is having a baby with %d tickets!\n", getpid(), num);
  if(fork(num) == 0){
    timewaster();
    printf(1, "Process %d is over. Number of Tickets: %d\n", getpid(), num);
    exit();
  }
}
for(i = 0; i < NPROC_T; i++) wait(); //Wait for Children to exit
```

Figura 2. Trecho do código do programa de teste onde os processos filhos são criados.

4. Conclusões

A implementação do escalonador por passos largos foi bem sucedida. O escalonador se mostrou eficiente e coerente com o descrito por seu criador. Algumas otimizações ainda poderiam ter sido feitas, como um mecanismo de controle global para que processos que foram inseridos uma determinada quantidade de tempo depois que outros, sejam escalonados de maneira mais justa, como o própio Waldspurger (criador do escalonamento por passos largos) propõe em seu artigo. [2]

Referências

[1] ”Xv6, a simple Unix-like teaching operating system”. https://pdos.csail.mit.edu/6.828/2012/xv6.html. Acessado em Maio de 2017.

[2] Weihl E. William, Waldspurger A. Carl. Lottery Scheduling: Flexible Proportional-Share Resource Management [*Cambridge: MIT Laboratory for Computer Science, Massachusetts Institute of Technology.*].

[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990] [*Introduction to Algorithms (3rd ed.)*].