Implementação do escalonador stride

Everton de Assis Vieira[1], **Sabrina Moczulski**[1]

[1]Curso de Ciência da Computação – Universidade Federal da Fronteira Sul (UFFS)
Chapecó – SC – Brazil

eassis.vieira@gmail.com.br, sabrina.moczulski@gmail.com

Abstract. *This paper describes and explains the planning and implementation of a process scheduler like stride in the xv6 operating system. It will describe the operating system analysis, the scheduler implementation, and the tests performed.*

Keywords– xv6, scheduling, stride

Resumo. *Este artigo descreve e explica o planejamento e a implementação de um escalonador de processos do tipo stride no sistema operacional xv6. Será descrita a análise do sistema operacional, a implantação do escalonador e os testes realizados.*

Palavas-chave– xv6, escalonador, passadas

1. Introdução

O xv6 é um sistema operacional didático desenvolvido pelo Massachusetts Institute of Technology (MIT) em 2006 por alunos da disciplina de sistemas operacionais.

Com o xv6 é possível aprender mais sobre a importância e as funções de sistemas operacionais. Implementar um escalonador permite a maior compreensão dos assuntos já estudados e possibilita que seja feita a comparação entre eles, os benefícios de cada um e as principais diferenças a nível de implementação.

Este trabalho foi realizado para disciplina de Sistemas Operacionais e auxilia na aprendizagem do que foi visto em sala de aula, possibilitando a visualização prática de vários conceitos estudados.

2. Stride Scheduling

Um escalonador de processos tem como função principal, escolher os processos que devem ser executados naquele momento. Existem algumas formas de escolher qual processo será executado. Neste artigo, trataremos sobre a utilização de um algoritmo determinístico para encontrar o processo que deve ser executado.

O escalonador Stride Scheduling[Waldspurger and Weihl. W. 1995] implementa o conceito de escalonamento por passadas. Este escalonamento consiste em atribuir um determinado valor de passo (stride) para cada processo, o passo é calculado através da divisão de um valor constante pelo número de bilhetes (tickets) do processo, esse valor será somado então ao valor da passada (pass) do processo a cada vez que o processo é escolhido pelo escalonador.

Para escolha do processo a ser executado, o escalonador irá selecionar o processo com menor passada.

3. Min-heap

A min-heap é uma árvore binária que é armazenada em um vetor. A implementação desse algoritmo permite obter o menor valor da árvore de forma eficiente, pois esse valor é sempre armazenado na raiz da árvore.

Através da realização das seguintes operações é possível navegar pelo vetor como se estivéssemos utilizando uma árvore:

```
parent = (i / 2)
left = (i * 2)
right = (i * 2 + 1)
```

As operações da min-heap vão sempre manter a propriedade de ter o pai de cada nodo menor que o filho. Desta forma, como mencionado anteriormente o menor valor estará sempre na raiz da árvore. E a remoção ou inserção de qualquer novo nodo na árvore tem complexidade de *O(log N)*.

4. Alterações na xv6

Todas as chamadas de sistema fork foram alteradas para que fosse atribuído um número de bilhetes como parâmetro da função.

Para a implementação do escalonador foram alterados os arquivos proc.h, proc.c e sysproc.c.

No arquivo *proc.h* foi alterada a estrutura do processo para que ele armazene a quantidade de bilhetes, bem como a passada e o passo do processo.

É importante saber que o xv6 possui 6 estados disponíveis para o processo: UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING e ZOMBIE pois é necessário saber quais devem ser utilizados durante a implementação do escalonador.

No arquivo *proc.c* foi implementada uma min-heap, com o objetivo de obter o processo com menor valor de passada de forma mais eficiente, já que para retirar o menor valor da min-heap a complexidade é *O(log N)* e para inserção também. Além disso, foi alterado o escalonador padrão da xv6 para um implementação do Stride Scheduling[Waldspurger and Weihl. W. 1995].

No arquivo *sysproc.c* foi alterada a chamada de sistema fork, incluindo na chamada o valor de bilhetes do processo.

5. Implementação do escalonador

A implementação da min-heap foi feita da seguinte forma:

```
struct {
  struct proc *heap[HMAX];
  int size;
} min_heap;
```

A estrutura da min-heap é bastante simples. Possui apenas um vetor com ponteiros para os processos e um valor do tipo inteiro para armazenar o tamanho da árvore, ou seja, quantos processos foram incluídos na árvore.

```
void
heap_push(struct proc *proc)
```

```
{
  int i = ++min_heap.size;
  struct proc *child, *parent;

  min_heap.heap[i] = proc;

  while(i > 1){
    child = min_heap.heap[i];
    parent = min_heap.heap[PARENT(i)];
    if(child->pass < parent->pass){
      min_heap.heap[i] = parent;
      min_heap.heap[PARENT(i)] = child;
    }
    --i;
  }

}
```

A função *heap_push* insere um processo na min-heap. Primeiramente é incrementado o valor do tamanho da árvore e incluído o processo no fim do vetor. Depois de incluído o processo no vetor é feito uma verificação das propriedades da min-heap, para que o pai dos nodos sempre tenha valor de passada menor que os filhos.

É importante dizer que são inseridos na min-heap apenas processos com estado RUNNABLE. Isso facilita a sua utilização no escalonador.

```
struct proc*
heap_pop()
{
  int i = 1, j;
  struct proc *child, *parent, *temp, *proc;

  if(!min_heap.size){ return 0; }

  proc = min_heap.heap[1];
  min_heap.heap[1] = min_heap.heap[min_heap.size--];

  while(i < min_heap.size){
    parent = min_heap.heap[i];
    child = 0;
    j = LEFT(i);
    if(LEFT(i) <= min_heap.size){
      child = min_heap.heap[j];
      if(RIGHT(i) <= min_heap.size){
        temp = min_heap.heap[RIGHT(i)];
        if(temp->pass < child->pass){
          j = RIGHT(i);
          child = min_heap.heap[j];
        }
      }
    }
```

```
    if(child){
      if(child->pass < parent->pass){
        min_heap.heap[i] = child;
        min_heap.heap[j] = parent;
      }
    }
    i = j;
  }

  return proc;
}
```

A função *heap_pop* faz a remoção do processo com menor passada e armazena esse processo na variável *proc, em seguida o último processo do vetor é inserido na posição do processo que foi removido anteriormente. Depois disso, ela realiza uma verificação semelhante a da *heap_push*, com o intuito de manter as propriedades da min-heap. Por último a função retorna o processo armazenado na variável *proc.

```
void
init_proc(struct proc *p, int tickets)
{
  p->tickets = tickets;
  p->stride = stride / p->tickets;
  p->pass = min_pass;
}
```

O processo é inicializado com a função *init_proc*, a variável stride é uma variável global utilizada para calcular o passo do processo. E a variável min_pass armazena o valor da passada do processo com menor passada. O valor de passada do processo sempre inicia com valor da variável min_pass.

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
        c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    p = heap_pop();
    if(p){
      if(p->state == RUNNABLE){
        c->proc = p;
        switchuvm(p);
        min_pass = p->pass;
        p->exec_times++;
        p->pass += p->stride;
```

```
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;
      }
    }
    release(&ptable.lock);
  }

}
```

Após a implementação da min-heap para a seleção do processo com o menor passo, implementar o escalonador de processos é uma tarefa simples. Basta retirar da min-heap o processo com a menor passada, fazer uma verificação para garantir que o estado do processo é RUNNABLE, salvar o valor de passada do processo na variável min_pass, somar o valor do passo do processo ao valor de passada e incrementar o atributo exec_times do processo para facilitar a visualização do número de vezes que o processo foi executado.

Além disso as chamadas *switchuvm* e *swtch* irão carregar a tabela de páginas do processo na memória, irá carregar o processo para ser executado, respectivamente. Após o término do quantum do processo, o escalonador será novamente carregado para execução e sua tabela de páginas também será carregada novamente na memória através da chamada *switchkvm*.

A inserção de processos na min-heap é feita nas funções *userinit fork*, *wakeup1*, *kill* e *yield*, que são as funções onde ocorrem a mudança do estado do processo para RUNNABLE.

6. Testes

Para verificar o funcionamento do escalonador stride foi criado um arquivo chamado de *procreator.c* para a realização dos testes.

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fs.h"

int main(int argc, char *argv[]){
  int qty_process = 0;
  int tickets[64];
  int id;

  if(argc == 1){
    printf(1, "Wrong format!\n");
    printf(1, "qty_process tickets tickets tickets")
    exit();
  }

  qty_process = atoi(argv[1]);

  for(int i = 0; i < qty_process; i++){
    process[i] = atoi(argv[i+2]);
  }
```

```
  for(int i = 0; i < qty_process; i++){
    printf(1, "Process %d tickets: %d\n", i, process[i]);
    id = fork(process[i]);
    if(id == 0){
      for(;;);
      exit();
    }
    if(id == -1) break;
  }

  for(int i = 0; i < qty_process; i++){
    wait();
  }

  exit();
}
```

O número de processos a ser criado é passado por argumento, bem como as suas respectivas quantidades de tickets. Então, são criados processos que irão executar em loop infinito. Isso possibilita visualizar o funcionamento do escalonador implementado anteriormente.

Os testes executados apresentaram os seguintes resultados:

```
$  sleep  init tckts: 50 times: 13 stride: 20971 pass: 293594 80104137 801041e1 80104bf7 80105c89 801059ef
2 sleep  sh tckts: 50 times: 23 stride: 20971 pass: 754956 801040fc 801002c2 80100f8c 80104ef2 80104bf7 80105c89 801059ef
5 runble procreator tckts: 50 times: 97 stride: 20971 pass: 2873027
4 sleep  procreator tckts: 50 times: 13 stride: 20971 pass: 922724 80104137 801041e1 80104bf7 80105c89 801059ef
6 runble procreator tckts: 100 times: 193 stride: 10485 pass: 2883416
7 runble procreator tckts: 200 times: 382 stride: 5242 pass: 2883226
8 run    procreator tckts: 400 times: 755 stride: 2621 pass: 2880608
9 run    procreator tckts: 500 times: 938 stride: 2097 pass: 2868739
```

Figura 1. Execução do teste

```
$  sleep  init tckts: 50 times: 13 stride: 20971 pass: 293594 80104137 801041e1 80104bf7 80105c89 801059ef
2 sleep  sh tckts: 50 times: 45 stride: 20971 pass: 1216318 801040fc 801002c2 80100f8c 80104ef2 80104bf7 80105c89 801059ef
5 runble procreator tckts: 50 times: 606 stride: 20971 pass: 13547266
4 sleep  procreator tckts: 50 times: 13 stride: 20971 pass: 922724 80104137 801041e1 80104bf7 80105c89 801059ef
6 runble procreator tckts: 100 times: 1210 stride: 10485 pass: 13546661
7 run    procreator tckts: 200 times: 2417 stride: 5242 pass: 13550696
8 runble procreator tckts: 400 times: 4824 stride: 2621 pass: 13545457
9 run    procreator tckts: 500 times: 6029 stride: 2097 pass: 13544566
```

Figura 2. Execução do teste

```
$  sleep  init tckts: 50 times: 13 stride: 20971 pass: 293594 80104137 801041e1 80104bf7 80105c89 801059ef
2 sleep  sh tckts: 50 times: 201 stride: 20971 pass: 4487794 801040fc 801002c2 80100f8c 80104ef2 80104bf7 80105c89 801059ef
5 runble procreator tckts: 50 times: 3022 stride: 20971 pass: 64213202
4 sleep  procreator tckts: 50 times: 13 stride: 20971 pass: 922724 80104137 801041e1 80104bf7 80105c89 801059ef
6 run    procreator tckts: 100 times: 6042 stride: 10485 pass: 64210181
7 runble procreator tckts: 200 times: 12080 stride: 5242 pass: 64204142
8 runble procreator tckts: 400 times: 24152 stride: 2621 pass: 64204145
9 run    procreator tckts: 500 times: 30188 stride: 2097 pass: 64205989
```

Figura 3. Execução do teste

Os processos criados pelo programa *procreator* são os processos com pids 5, 6, 7, 8 e 9, que foram criados com 50, 100, 200, 400 e 500 bilhetes respectivamente.

Como podemos observar a proporcionalidade é mantida, os processos com maior número de bilhetes são escolhidos mais vezes. Observando bem, o processo 6 que possui o dobro de bilhetes em relação ao processo 5 estão executando o dobro de vezes, e assim sucessivamente.

7. Conclusões

Com base nos testes realizados, podemos concluir que foi possível implementar o escalonador stride com sucesso. A maior dificuldade encontrada durante a execução do trabalho proposto foi a implementação da min-heap.

Com este trabalho foi possível também a compreensão de uma forma determinística de escalonamento, que se mostra mais correta quando falamos de compartilhamento de CPU. Além disso, o *stride scheduling*, quando bem implementado, não deixa tanta margem para erros como o *lottery scheduling*.

Referências

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