Implementação de escalonamento por loteria no *xv6* com análise de desempenho

João Pedro Winckler Bernardi[1]

[1]Universidade Federal da Fronteira Sull (UFFS)
89802-260 – Chapecó – SC – Brasil

jpwbernardi@hotmail.com

Abstract. *In a lottery scheduler, for each process is given a certain amount of tickets. A random ticket is selected and the process that owns that ticket is executed. This report shows the lottery sceduler's BIT – Binary Indexed Tree – implementation on xv6 and analyzes if works as required.*

Resumo. *No escalonamento por loteria, cada processo recebe uma quantidade de bilhetes. O processo com o bilhete sorteado é executado. Nesse relatório, apresentamos uma implementação feita com uma BIT – Binary Indexed Tree – para o escalonador por loteria no sistema operacional xv6 e analisamos seu funcionamento.*

1. Introdução

Os sistemas operacionais executam vários processos ao mesmo tempo, ou, pelo menos, é essa a impressão que temos. Um CPU não consegue executar mais de um processo simultaneamente, o que ocorre é que todo processo está competindo para ganhar fatias de tempo do processador. O escalonador é o responsável pela escolha do processo a ser executado. Na troca de processos, o estado do processo atual deve ser salvo para que na próxima vez que ele for escolhido esses dados sejam recarregados e ele possa continuar a execução exatamente do momento que parou [Tanenbaum 2010].

O método que o escalonador vai usar para escolher o próximo processo é chamado de algoritmo de escalonamento. Esse relatório apresenta o funcionamento do escalonamento por loteria com complexidade de $O(\log n)$, sendo n a quantidade máxima de processos, e sua implementação no sistema operacional xv6.

2. Escalonamento por loteria

Esse algoritmo de escalonamento funciona da seguinte forma: cada processo recebe uma quantidade de bilhetes, então sorteia-se um bilhete e o processo dono daquele bilhete é executado. Embora todos os processos tenham bilhetes, só os processos prontos(*RUNNABLE* no xv6) devem ser sorteados. Assim surgiu a ideia do acumulo de bilhetes.

Associamos cada processo a uma posição i de um vetor que guarda a soma de bilhetes do processo na posição 0 até a posição i. Por exemplo, os processos A, B, C, D, E com 10, 20, 5, 1 e 13 bilhetes, respectivamente, estão prontos. Por tanto, temos 49 bilhetes que podem ser sorteados e nosso vetor estaria da seguinte forma:

Tabela 1. Exemplo do método de bilhetes acumulados 1

A	B	C	D	E
10	30	35	36	49

Isso significa que os bilhetes de A estão entre 1 e 10, os de B entre 11 e 30, os de C entre 31 e 35, o de D é 36 e os de E entre 37 e 49. Então, suponhamos que o bilhete sorteado foi o 15, ou seja, o processo B será executado. Mas durante sua execução foi bloqueado, isso implica que B não pode mais ser executado enquanto não estiver *RUNNABLE*. Portanto, no vetor onde guardamos os acúmulos, retiramos os bilhetes de B. O vetor de acúmulos estaria da seguinte maneira:

Tabela 2. Exemplo do método de bilhetes acumulados 2

A	B	C	D	E
10	10	15	16	29

Agora, os bilhetes de A estão entre 1 e 10, B não tem bilhete, os bilhetes de C estão entre 11 e 15, o de D é 16 e os de E entre 17 e 29. Note que é apenas no vetor de acúmulos que os bilhetes são retirados, a informação que o B tem 20 bilhetes continua guardada, pois quando ele estiver pronto novamente, devolveremos os bilhetes para o vetor de acumulos.

Trabalhando com o acumulo de bilhetes, fica simples de se implementar o sorteio de um bilhete, pois é um intervalo contínuo. Se tivéssemos atribuído números fixos aos bilhetes, quando um processo fosse bloqueado geraria uma buraco no intervalo de bilhetes a serem sorteados e teríamos que tratar para o sorteio não considerar esses buracos.

Assim, chega-se a outro problema: fazer o acúmulo de bilhetes de forma eficiente. Pois, como apresentado até agora, a ideia simples para fazer o cálculo desses acumulos seria percorrendo o vetor e atualizando cada posição, uma complexidade de $O(n)$, sendo n a quantidade máxima de processos permitidos, para cada vez que fosse necessário atualizar a quantidade de bilhetes um processo. Porém, existe uma estrutura de dados chamada BIT – Binary Indexed Tree – que faz essa operação com complexidade $O(\log n)$, conforme [Halim and Halim 2013].

Por fim, o último método utilizado para manter a complexidade baixa é buscar o processo com o bilhete sorteado através de uma busca binária. Portanto, o processo que adotamos é: sorteia um bilhete dentre os disponíveis, descobre que processo é dono desse bilhete através da busca binária, descrita a seguir.

```
int bsearch(int ticket){
  int l = 1, h = NPROC, m;
  while (l < h) {
      m = l + (h - l) / 2;
      if (ticount(m) >= ticket) h = m;
      else l = m + 1;
  }
  return l - 1;
}
```

3. Binary Indexed Tree(BIT)

Inventada por Peter M. Fenwick em 1994, a BIT é uma estrutura de dados simples para implementar tabelas de frequências cumulativas. A implementação foi feita através de um vetor, onde cada posição guarda um acumulo parcial. Na implementação, a BIT é representada pelo vetor $stickets$. O vetor $idstack$ é uma pilha estática que guarda as posições da BIT não associadas a processos, tp é em que posição se encontra o topo da pilha. É importante dizer que a posição 0 da BIT não é utilizada.

```
struct {
  int stickets[NPROC + 1];
  int idstack[NPROC], tp;
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

De uma forma mais genérica, o elemento da posição i da BIT $stickets$ é responsável pelos elementos no intervalo [$i - (i$ AND $-i) + 1$, i] e $stickets[i]$ guarda o acumulo dos bilhetes $\{i - (i$ AND $-i) + 1, i - (i$ AND $-i) + 2, i - (i$ AND $-i) + 3$, ... , i$\}$.

Para obtermos o acúmulo de bilhetes de uma posição, usamos a função *ticount*. Para atualizarmos uma posição da BIT com algum valor, usamos a função *uptick*.

```
int ticount(int i){
  int count = 0;
  for (; i > 0; i -= i & -i)
    count += ptable.stickets[i];
  return count;
}

void uptick(int i, int value){
  for (; i <= NPROC; i += i & -i)
    ptable.stickets[i] += value;
}
```

4. Mudanças no *xv6*

Inicialmente, o pid de um processo era definido por uma variável global. Toda vez que um processo recebia um pid, a global era incrementada. Porém, para facilitar a implementação do código e sua otimização, agora o pid representa o índice da BIT associado ao processo. Toda vez que um processo termina, ele devolve seu pid para a pilha $idstack$. Como cada processo tem um pid único que varia de 1 a n, então também podemos usar o pid para a indexação do processo no vetor de processos $ptable.proc$, obviamente subtraindo 1, pois as posições de $ptable.proc$ são de 0 a $n - 1$. Antes, quando um processo era criado, para achar uma posição livre no vetor de processos, este era percorrido linearmente. Agora, só retiramos da pilha um pid disponível, o que é realizado em tempo $O(1)$.

```
static struct proc* allocproc(void){
```

```
 ...
  acquire(&ptable.lock);
  if (ptable.tp > 0) {
    i = ptable.idstack[--ptable.tp];
    p = &ptable.proc[i - 1];
    goto found;
  }
  release(&ptable.lock);
  return 0;
found:
  p->pid = i;
 ...
  return p;
}
```

Toda vez que o *xv6* é inicializado, a função $clean$ é chamada. Ela é responsável por inicializar a pilha, ou seja, colocar todas as posições disponíveis da BIT na pilha, e zerar a BIT.

```
void clean() {
  acquire(&ptable.lock);
  for (ptable.tp = 0; ptable.tp < NPROC; ptable.tp++)
    ptable.idstack[ptable.tp] = NPROC - ptable.tp;
  memset(ptable.stickets, 0, sizeof (ptable.stickets));
  release(&ptable.lock);
}
```

E a função *scheduler* é a responsável pela mudança de processo em execução. Primeiro verificamos a quantidade de bilhetes disponíveis para serem soreados. Se essa quantidade for diferente de 0, ou seja, houver algum bilhete para ser sorteado, sorteamos um bilhete entre 1 e a quantidade de bilhetes. A busca binária retorna a posição no vetor *ptable.proc* do processo que tinha o bilhete sorteado. Então, retiramos os bilhetes desse processo e mudamos seu estado para *RUNNING*.

```
void scheduler(void){
  ...
  for(;;){
    sti();
    acquire(&ptable.lock);
    if ((qttytickets = ticount(NPROC)) != 0) {
      p = &ptable.proc[bsearch(rand() % qttytickets + 1)];
      if (p->state == RUNNABLE) {
        uptick(p->pid, -p->tickets);
        proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&cpu->scheduler, proc->context);
        switchkvm();
        proc = 0;
```

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

Os bilhetes do processo são devolvidos quando o tempo dele no CPU termina e ele volta para o estado *RUNNABLE*.

```
void yield(void){
  acquire(&ptable.lock);
  proc->state = RUNNABLE;
  uptick(proc->pid, proc->tickets);
  sched();
  release(&ptable.lock);
}
```

5. Análise de desempenho

Toda vez que o escalonador busca um processo para ser executado, é chamado a busca binária, que tem complexidade $O(\log n)$, porém, para acessar o valor do acúmulo numa posição i, a complexidade é também $O(\log n)$ [Halim and Halim 2013]. Ou seja, a complexidade total para escolher um processo seria $O(\log(n \cdot \log n))$. Desenvolvendo a conta:

$$O(\log(n \cdot \log n)) = O(\log n + \log(\log n)) = O(\log n)$$

6. Análise de testes

Para testar o funcionamento do que foi implementado, criamos um arquivo $schedtest$ no $xv6$. Quando executado, cria o máximo de processos possíveis. Cada processo decrementa uma variável que começa em aproximadamente 10^8 e, quando essa variável chega a 0, o processo acaba. Para obter os resultados do teste, foi modificado a função $exit()$ para que quando ela fosse chamada, exibisse a quantidade de tickets do processo que acabou. Cada processo tem uma quantidade de bilhetes diferentes. Esse teste foi realizado 10 vezes com processos de mesma quantidade de bilhete. A quantidade de bilhetes de cada processo é $n^{o}doprocesso \cdot 64 + 1$.

O resultado obtido está apresentado na Tabela 3. Cada coluna representa um teste e cada linha representa a ordem que o processo acabou. Por exemplo, no teste 1, o processo 38 foi o 6º processo a terminar e no teste 7 foi o 27º.

Como podemos observar, um processo ter mais bilhetes não é garantia de que ele vai ganhar mais tempo no processador. Possívelmente pela pouca diferença de bilhetes entre cada processo e pelo $rand$ que foi implementado para os testes, isso se tornou mais evidente. Um exemplo disso é o processo 60 que foi o 26º a terminar no teste 4, embora possuísse mais bilhetes que qualquer outro processo, e nunca terminou por primeiro.

Vale ressaltar que, embora aconteça o descrito acima, os processos com mais bilhetes ainda tendem a terminar antes, mas por ser um algoritmo probabilístico, não temos como dizer que isso sempre ocorre.

Teste 1	Teste 2	Teste 3	Teste 4	Teste 5	Teste 6	Teste 7	Teste 8	Teste 9	Teste 10
54	49	36	32	44	42	50	38	36	39
40	51	58	46	41	55	53	36	52	42
59	52	59	42	45	59	58	40	45	48
47	41	44	44	52	37	44	33	42	38
44	55	51	57	50	57	48	44	37	50
38	38	53	29	46	53	52	42	47	45
49	46	35	45	42	33	49	43	41	51
48	44	55	59	48	49	42	28	59	55
35	57	48	36	38	44	37	45	38	47
50	43	39	38	43	58	56	37	32	52
60	60	54	34	49	47	60	48	40	53
52	58	38	40	54	46	45	55	51	58
32	39	47	35	47	40	41	30	55	43
42	50	60	27	58	51	59	47	58	41
55	48	43	52	53	45	34	60	39	49
57	37	45	41	37	48	46	58	60	37
41	45	41	50	59	43	54	50	49	31
58	59	50	51	40	41	51	39	31	27
51	35	46	31	31	32	40	49	50	46
37	47	29	47	56	36	55	27	44	40
45	54	42	53	55	50	38	53	56	30
53	33	26	37	57	34	30	32	53	44
56	56	56	54	60	30	36	54	54	54
36	42	24	39	34	56	57	35	29	33
33	40	57	30	36	60	39	52	57	57
30	32	34	60	39	54	43	56	43	35
43	53	33	58	23	39	47	29	27	60
39	34	49	43	28	52	35	41	48	32
31	26	52	26	51	25	25	57	46	36
29	30	23	55	29	35	26	34	33	56
46	29	40	23	35	26	29	46	34	25
28	28	30	48	27	31	31	51	25	34
21	27	37	49	19	38	32	59	28	59
34	36	32	25	26	23	33	31	24	26
22	24	28	56	24	29	27	21	35	21
27	25	19	33	33	24	21	25	22	28
26	31	31	21	32	18	28	26	23	20
23	23	22	28	22	28	23	23	26	23
25	22	25	17	30	27	24	22	30	24
20	21	27	22	25	20	22	24	21	29
24	19	14	24	18	21	19	19	19	17
19	14	16	19	21	13	20	17	18	19
18	20	15	20	15	17	18	20	20	22
15	18	20	18	20	22	15	18	17	14
17	16	18	16	16	19	17	16	15	18
16	15	17	15	13	16	16	15	16	15
13	13	21	11	17	15	14	12	14	16
14	12	13	13	14	14	12	11	13	12
11	17	11	10	12	12	13	14	12	13
10	8	12	12	11	11	10	13	10	10
12	11	9	14	9	10	11	10	11	9
9	9	10	8	8	9	8	8	9	11
8	10	8	7	7	8	7	9	8	8
6	5	7	9	10	6	9	7	6	7
7	7	6	6	6	7	6	5	7	6
5	6	5	5	5	5	5	6	5	5
4	4	4	4	4	4	4	4	4	4
3	3	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	1	1

Tabela 3. Resultados

Referências

Halim, S. and Halim, F. (2013). *Competitive Programming 3*.

Tanenbaum, A. S. (2010). *Sistemas Operacionais Modernos*. Pearson Prentice Hall, 3rd

edition.