Implementação de um Escalonador por Loteria no Xv6

Igor Lemos Vicente[1], **Leonardo Vargas**[1]

[1]Universidade Federal da Fronteira Sul
89812-000 – Chapecó – SC – Brazil

igorlemosvicente@gmail.com, leu1607@hotmail.com

Abstract. *This paper shows the steps taken implementing a simple version of the Lottery Scheduling algorithm on xv6. The used version uses a pseudorandom lottery and does not use some concepts presents in the original algorithm, such as transfer, inflaction and compensation.*

Resumo. *Este artigo mostra os passos tomados na implementação de uma versão simples do algoritmo de Escalonamento por Loteria no xv6. A versão utilizada possui um sorteio pseudoaleatório e não utiliza alguns conceitos do algoritmo original, como transferência, inflação ou compensação.*

1. Entendendo o XV6

1.1. Sobre

O XV6 é um sistema operacional didático desenvolvido no ano de 2006 pelo MIT (Massachusetts Institute of Technology) no curso de sistemas operacionais. Ele foi inspirado pela sexta edição do UNIX (aka V6) para a plataforma Intel x86.
O x86 permitiu a unificação do desenvolvimento sobre uma única arquitetura, além de oferecer o suporte multiprocessador.

1.2. Processo

Como visto em [Cox et al. 2017] cada processo criado no XV6 detém um dos seis tipos de estados, que são: SLEEPING, RUNNING, RUNNABLE, UNUSED, ZOMBIE ou EMBRYO. Cada estado só pode ser alterado quando o processo adentrar na zona crítica. Assim o Kernel do sistema é o responsável por controlar o tempo que cada processo utilizará da CPU.

Estrutura de um processo no XV6 (**proc.h**):

```
1  struct proc {
2    uint sz;                     // Size of process memory (bytes)
3    pde_t* pgdir;                // Page table
4    char *kstack;                // Bottom of kernel stack for
       this process
5    enum procstate state;        // Process state
6    int pid;                     // Process ID
7    struct proc *parent;         // Parent process
8    struct trapframe *tf;        // Trap frame for current syscall
9    struct context *context;     // swtch() here to run process
10   void *chan;                  // If non-zero, sleeping on chan
```

```
11    int killed;                  // If non-zero, have been killed
12    struct file *ofile[NOFILE];  // Open files
13    struct inode *cwd;           // Current directory
14    char name[16];               // Process name (debugging)
15  };
```

1.3. Escalonador

No escalonador nativo do XV6 é criado um vetor onde se têm todos processos do sistema. O critério de escolha é simples, o escalonador faz uma varredura neste vetor e tenta encontrar o primeiro processo que esteja no estado RUNNABLE. Fragmento de código sobre o escalonador (**proc.c**):

```
1  void scheduler(void){
2    struct proc *p;
3    struct cpu *c = mycpu();
4    c->proc = 0;
5    for(;;){
6      sti();
7      acquire(&ptable.lock);
8      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
9        if(p->state != RUNNABLE)
10         continue;
11       c->proc = p;
12       switchuvm(p);
13       p->state = RUNNING;
14       swtch(&(c->scheduler), p->context);
15       switchkvm();
16       c->proc = 0;
17     }
18     release(&ptable.lock);
19   }
20 }
```

2. Algoritmo de Escalonamento por Loteria

No escalonador de processos baseado em loteria (lottery scheduling), cada processo possui um número de bilhetes (tickets), com isso o sistema fará um sorteio onde o processo que deter o número sorteado será o "ganhador" sendo selecionado pelo escalonador, como visto em [Waldspurger and Weihl 1994].

3. Mudanças na estrutura

3.1. Processo

Manteve-se a estrutura do processo, mas foram adicionados alguns campos. O campo *tickets* é onde se armazena a quantidade de tickets que um processo detém e os campos *tickets_soma, escolhido e cogitado* foram criados para guardar as informações referentes aos ciclos de escalonamento de cada processo.
Campos adicionados no arquivo **proc.h**:

```
1  struct proc {
2    ...
3    int tickets;       // Numero de tickets do processo
4    int tickets_soma;  // Variavel para guardar a soma da soma de
       tickets distribuidos nas escolhas que o processo participou
5    int escolhido;     // Numero de vezes que o processo foi
       escolhido
6    int cogitado;      // Numero de vezes que o processo poderia
       ter sido escolhido
7    ...
8  };
```

3.2. Função para números pseudoaleatórios

```
1  unsigned long randstate = 1;
2  unsigned int rand() {
3    randstate = randstate * 1664525 + 1013904223;
4    return randstate < 0 ? randstate * -1 : randstate;
5  }
```

A função **rand** fora criada para gerar números pseudoaleatório para fins de testes da implementação do escalonamento. Foi utilizada uma função existente no arquivo de testes do xv6 (**usertests.c**) e alterado o retorno da função para números positivos apenas. Acima a função alterada.

3.3. Mudança na função de alocação de processo

Em **allocproc**, função responsável pela alocação do processo na memória, fez-se necessário a criação de um argumento como entrada com o número de tickets que o processo possuirá.

```
1  static struct proc* allocproc(int tickets_number){
2    ...
3    p->state = EMBRYO;
4    p->pid = nextpid++;
5    tickets_number %= 1000;        // Um processo nao vai possuir
       mais que 1000 tickets
6    p->tickets = tickets_number ? tickets_number :
       DEFAULT_TICKETS_NUMBER;
7    p->tickets_soma = 0;
8    p->escolhido = 0;
9    p->cogitado = 0;
10   release(&ptable.lock);
11   ...
12 };
```

Foram alteradas também as chamadas da função em **userinit** e **fork**, para adequar ao novo cabeçalho da função de alocação, passando como argumento um número de tickets pseudoaleatório.

```
1  int fork(void){
2    ...
```

```
3     // Allocate process.
4     if((np = allocproc(rand())) == 0)
5       return -1;
6     ...
7   };
8
9   void userinit(void){
10    ...
11    p = allocproc(rand());
12    ...
13  };
```

3.4. Escalonador

Agora com todas a modificações feitas, implementamos o escalonador por loteria. Onde o escalonador percorre todos os processos e adiciona aqueles que estão no estado RUNNABLE a um vetor secundário denominado *lista_runnable*. Enquanto o vetor de processos é percorrido uma variável é responsável por armazenar o somatório total de tickets daquele vetor. Ela será utilizada no sorteio, pois com ela pode-se limitar o intervalo do número sorteado.

Logo após, percorre-se todos os processos do vetor de "RUNNABLES" verificando qual processo é dono do bilhete sorteado. Ao encontrar o processo o escalonador entra na zona crítica onde é mudado o estado do processo para RUNNING e é feita a mudança de contexto da CPU para o contexto do processo. Depois disso o escalonador sai da zona crítica e atualiza a tabela de processos. Durante este processo são atualizadas as informações que posteriormente serão utilizadas para a apresentação dos resultados mostrados no presente artigo.

```
1   void scheduler(void) {
2     struct proc *p;
3     struct proc *aux;
4     struct cpu *c = mycpu();
5     c->proc = 0;
6     struct proc* lista_runnable[NPROC];
7     int indice_lista_runnables, soma_tickets, i, ticket_sorteado,
        tickets_passados;
8     for(;;){
9       sti();
10      acquire(&ptable.lock);
11      indice_lista_runnables = 0;
12      soma_tickets = 0;
13      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
14        if (p->state == RUNNABLE) {
15          lista_runnable[indice_lista_runnables++] = p;
16          soma_tickets += p->tickets;
17        }
18      }
19      if (soma_tickets > 0) {
20        ticket_sorteado = rand() % soma_tickets;
21        tickets_passados = 0;
```

```
22        for (i = 0; tickets_passados < ticket_sorteado; i++) {
23          p = lista_runnable[i];
24          tickets_passados += p->tickets;
25        }
26
27        // Atualizacao de Informacoes
28        for (aux = ptable.proc; aux < &ptable.proc[NPROC]; aux++)
             {
29          if (p->state == RUNNABLE) {
30            aux->tickets_soma += soma_tickets;
31            aux->escolhido += aux == p ? 1 : 0;
32            aux->cogitado += 1;
33          }
34        }
35        c->proc = p;
36        switchuvm(p);
37        p->state = RUNNING;
38        swtch(&(c->scheduler), p->context);
39        switchkvm();
40        c->proc = 0;
41      }
42      release(&ptable.lock);
43
44    }
45  }
```

3.5. Processo de Teste

O arquivo de teste **proc_teste.c** foi codificado da seguinte forma:

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  #define OUTPUT_STREAM 1
6
7  int main(void) {
8    int pid;
9    pid = fork();
10   if (pid) pid = fork();
11   if (pid) pid = fork();
12   if (pid) pid = fork();
13   if (pid) pid = fork();
14   if (pid) pid = fork();
15   if (pid) pid = fork();
16   if (pid) pid = fork();
17   if (!pid) {
18     int i = 0;
19     while(i++ < 2123) { // Whiles aninhados apenas para tomar
          tempo
```

```
20          int j = 0;
21          while(j++ < 1) {
22            int b = getpid() * 5 / 6 + 10;
23            b = b + b + getpid();
24          }
25        }
26    } else if (pid > 0) {
27        wait();
28    } else {
29        printf(OUTPUT_STREAM, "Erro\n");
30    }
31    exit();
32  }
```

A função única do processo de teste é criar 8 filhos e esperar que eles terminem de executar. A função dos filhos é executar o ciclo apenas para conseguir tempo de máquina suficiente para se tirar informações do escalonamento.

4. Transbordamento da pilha do kernel

Durante a implementação e execução dos testes, ao iniciar o xv6 com o emulador Qemu ou ao executar o processo de teste, erros de transbordamento da pilha do kernel aconteciam. Execuções consecutivas sem nenhuma alteração entre elas também exibiam o erro no terminal, sinalizando que o problema pode não estar conectado com as alterações feitos. Os transbordamentos aconteciam menos quando os números usados na função **rand** eram números não-negativos. Não foi encontrado o motivo para tais transbordamentos até o momento de finalização do presente artigo.

5. Ferramentas Utilizadas

Foi usada, para as etapas descritas neste artigo, as seguintes ferramentas:

- Sistemas Operacionais Ubuntu 17.04 64bit e Fedora 25 64bit
- Editor de texto Sublime3 e Atom para edição do código fonte
- Emulador Qemu como máquina virtual para execução do xv6

6. Resultados

A tabela 1 contém os resultados obtidos em três execuções de testes. Na primeira coluna é mostrado o número de tickets que o processo recebeu ao ser inicializado. Na segunda coluna é mostrado a soma da soma de tickets de todos os ciclos do escalonador em que o processo esteve presente. A terceira e a quarta coluna mostram o número de vezes que o processo pôde ser escolhido e a quantidade de vezes que ele foi escolhido, respectivamente. A quinta coluna contém a média de tickets por ciclo do escalonador (MTPCE). É a soma da segunda coluna sobre a quantidade de vezes que o processo foi cogitado. A sexta coluna mostra a percentagem esperada de escolha do processo pelo escalonador. É a porcentagem do número de tickets do processo em relação à MTPCE. A sétima e última coluna mostra a porcentagem alcançada de escolha do processo.

Tabela 1. Resultados dos Testes

Nº. de Tickets	Soma de Tickets	Cogitado	Escolhido	MTPCE	% Esperada	% Alcançada
364	139210	41	8	3395.37	10.72	19.51
896	130404	35	9	3725.83	24.05	25.71
978	148582	43	8	3455.40	28.30	18.60
554	158421	55	15	2880.38	19.23	27.27
880	157758	46	9	3429.52	25.66	19.57
1352	111918	45	9	2487.07	14.15	20
83	168554	68	9	2478.74	3.35	13.24
100	175017	71	10	2465.03	4.06	14.08
163	165547	75	10	2207.29	2.85	13.33
1955	101938	22	8	4633.55	20.61	36.36
792	160115	49	17	3267.65	24.24	34.69
695	164948	42	8	3927.33	17.70	19.05
880	185898	46	8	4041.26	21.78	17.39
884	182984	46	9	3977.91	22.22	19.57
609	215552	60	10	3592.53	16.95	16.67
307	226963	72	9	3152.26	9.74	12.5
262	226423	79	11	2866.11	9.14	13.92
186	201316	75	9	2684.21	3.20	12
695	102024	28	8	3643.71	19.07	28.57
939	108749	25	8	4349.96	21.59	32
543	132705	43	15	3086.16	17.59	34.88
983	170926	39	8	4382.72	22.43	20.51
1955	165550	38	9	4356.58	21.92	23.68
880	204844	56	11	3657.93	24.06	19.64
307	214970	72	9	2985.69	10.28	12.5
100	211025	80	12	2637.81	3.79	15
186	190514	75	9	2540.19	3.39	12

7. Conclusão

Embora um terço dos resultados tenham se aproximado da porcentagem esperada, os outros dois terços - portanto, a maioria - não foram satisfatórios. Acredita-se que a razão deste não tão preciso desfecho se dá nos problemas enfrentados pelo transbordamento do kernel, impossibilitando a criação de processos com maiores tempos de execução, e com isso dificultando a obtenção de números mais precisos.

Referências

Cox, R., Kaashoek, F., and Morris, R. (2017). xv6: a simple, unix-like teaching operating system.

Waldspurger, C. A. and Weihl, W. E. (1994). Lottery scheduling: flexible proportional-share resource management. In *OSDI '94 Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. USENIX Association Berkeley, CA, USA.