

Escalonador de processos baseado em loteria aplicado ao xv6

Bruno Ribeiro, Lucas Percisi

¹Ciência da Computação – Universidade Federal da Fronteira Sul – Campus Chapecó
CCR: Sistemas Operacionais

Abstract. *Implementation of the lottery-based process scheduler applied to the xv6 operating system. In the instantiation of a process, the amount of tickets that the new process receives is transferred to the system. If the user does not supply this data, the system assumes a default number of tickets. We also assume a maximum number of tickets a process can receive.*

Resumo. *Implementação do escalonador de processos baseado em loteria (lottery scheduling) aplicado ao sistema operacional xv6. Na instanciação de um processo, passa-se ao sistema a quantidade de bilhetes que o novo processo recebe. Caso o usuário não forneça esse dado, o sistema assume um número default de bilhetes. Assumimos também um número máximo de bilhetes que um processo pode receber.*

1. Introdução

Tem se tornado cada vez mais comum um sistema computacional ser capaz de gerenciar quantidades de processos maior que a quantidade de CPU's disponível. Dessa forma torna-se necessário o uso de escalonadores, sendo esses programas que gerenciam qual processo deve ser executado pela CPU, da mesma maneira, o escalonador deve ter estratégias, recursos e uma estrutura de dados que o permita escolher o processo de maneira inteligente, sendo capaz de prevenir deadlocks e outros problemas [de Oliveira et al. 2010]. No presente trabalho iremos implementar o escalonador por loteria no sistema operacional xv6.

2. Sobre o xv6 OS

O sistema operacional xv6 é uma reimplementação do Unix V6 de Dennis Ritchie e Ken Thompson, implementado para operar sobre a estrutura multiprocessador x86 (então do nome xv6), utilizando a linguagem ANSI C na maioria de sua estrutura. O xv6 OS foi desenvolvido no verão de 2006 para o curso de sistemas operacionais do MIT. Os arquivos do xv6 podem ser encontrados em <https://github.com/mit-pdos/xv6-public>.

3. Sobre o escalonador por loteria

O escalonador por loteria é um escalonador probabilístico, cada processo recebe um determinado número de fichas ao iniciar. A cada sorteio do escalonador, o processo que possuir a ficha sorteada é executado. Caso um processo necessite ter mais prioridade que outro, esse processo deve possuir mais fichas para o sorteio, aumentando assim a probabilidade de ser sorteado [Tenenbaum 2011].

3.1. Escalonador do xv6

O escalonador do xv6 encontra-se no arquivo `proc.c` e é implementado na função `void scheduler(void)`. Como sabemos, a maioria dos sistemas operacionais multi-processos possui um escalonador, e quando o mesmo inicia, ele entra em um loop infinito sem retorno, mas fica nesse loop administrando os processos para serem executados. No caso do xv6, ao entrar no loop infinito, é verificado na tabela de processos qual está disponível para ser executado, e então realiza-se a mudança de estado do processo e a troca de contexto na CPU (preempção). A Imagem 1 mostra o escalonador original do xv6:

```

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);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;

            // Switch to chosen process. It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchvm(p);
            p->state = RUNNING;

            swtch(&(c->scheduler), p->context);
            switchkvm();

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

```

Figura 1. Escalonador original

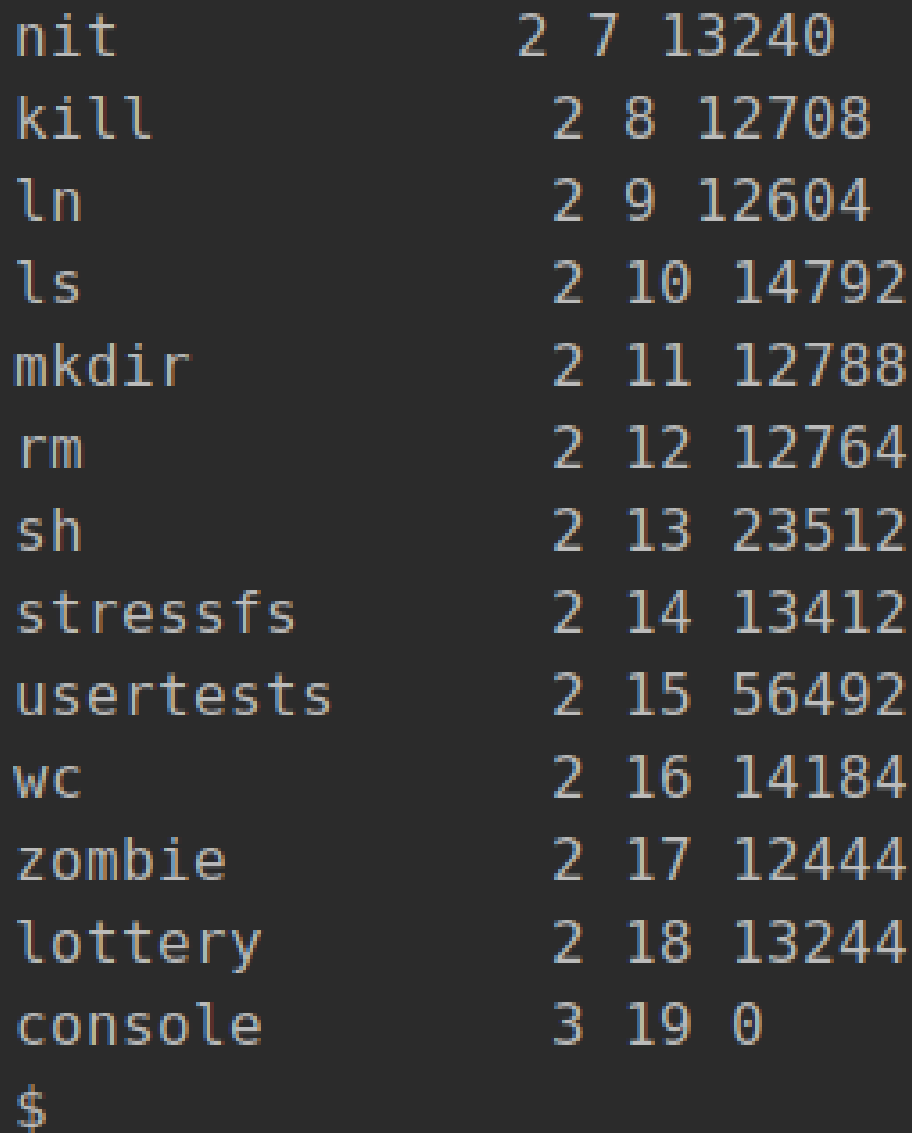
4. Reimplementação

O objetivo do presente trabalho é implementar o escalonador por loteria no xv6 OS. Para tanto iniciamos criando uma nova chamada de sistema chamada *lottery*, sendo que, essa chamada de sistema servirá simplesmente para testar o escalonador. A fim de verificar os efeitos do escalonador, após a chamada de sistema será impressa uma tabela

periodicamente com informações dos processos, uso real da CPU e uso estimado da CPU para cada processo a fim de comprovar seu funcionamento.

4.1. Chamada de Sistema

Para criar essa nova chamada de sistema alteramos os arquivos *Makefile*, *syscall.c*, *syscall.h*, *sysproc.c*, *user.h*, *usys.S*, para que, ao digitarmos no console do xv6 a chamada de sistema 'lottery', o mesmo execute o conteúdo do arquivo *lottery.c* (créditos ao **Siddharth Singh** por criar um post no blog <https://01siddharth.blogspot.com/2018/04/adding-system-call-in-xv6-os.html>, mostrando como adicionar uma chamada de sistema no xv6). Observamos na Imagem 2 a chamada de sistema criada.



```
nit          2  7  13240
kill         2  8  12708
ln           2  9  12604
ls           2 10  14792
mkdir        2 11  12788
rm           2 12  12764
sh           2 13  23512
stressfs     2 14  13412
usertests    2 15  56492
wc           2 16  14184
zombie       2 17  12444
lottery      2 18  13244
console      3 19   0
$
```

Figura 2. Comando 'ls' após criação da chamada sistema 'lottery'

4.2. Fork

Após a implementação da chamada de sistema, foi necessário alterar a função *int fork(void)* para *int fork(int)* pois dessa maneira, sempre que houver a instanciação de um novo processo, esse processo deve receber um número inteiro que representa a quantidade de bilhetes que o mesmo possui.

O arquivo *param.h*, possui macros do sistema que são utilizadas para configurá-lo. Neste arquivo foi adicionado a macro 'NTICKETS' que possui um valor padrão para instanciação de novos processos, quando estes não recebem o número de bilhetes. Esse valor é a razão do máximo de processos que o xv6 pode instanciar pela quantidade de cpu's (NPROC/NCPU). Também adicionamos a macro 'MAXTICKETS' que é usado para um teste dentro do *int fork(int)*, afim de que não exista processos com uma quantidade de bilhetes acima desse valor. A Imagem 3 mostra o teste:

```
181 // Create a new process copying p as the parent.
182 // Sets up stack to return as if from system call.
183 // Caller must set state of returned proc to RUNNABLE.
184 int
185 fork(int tickets)
186 {
187     int i, pid;
188     struct proc *np;
189     struct proc *curproc = myproc();
190
191     // Allocate process.
192     if((np = allocproc()) == 0){
193         return -1;
194     }
195
196     // Copy process state from proc.
197     if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
198         kfree(np->kstack);
199         np->kstack = 0;
200         np->state = UNUSED;
201         return -1;
202     }
203
204     // VERIFICA OS TICKETS PASSADO POR ARGUMENTO E DEFINE TETO
205     if (tickets <= 0) {
206         np->tickets = NTICKETS;
207     } else if (tickets > MAXTICKETS) {
208         np->tickets = MAXTICKETS;
209     } else {
210         np->tickets = tickets;
211     }
```

Figura 3. Testes para definir o máximo de bilhetes.

4.3. Escalonador

Abaixo podemos ver a implementação do nosso escalonador por loteria, as figuras 4, 5 e 6 mostram o algoritmo:

```
364 void scheduler(void){
365
366     struct proc *p;
367     struct cpu *c = mycpu();
368     int count = 0;
369     long golden_ticket = 0;
370     int total_no_tickets = 0;
371     int occurrences[NPROC];
372
373     for (int i = 0; i < NPROC; i++) {
374         occurrences[i] = 0;
375     }
376
377     c->proc = 0;
378
379     int d = 0;
380     total_no_tickets = tickets_total();
381     golden_ticket = random_at_most(ticks);
382
383     for(;;){
384
385         // Enable interrupts on this processor.
386         sti();
387
388         // Loop over process table looking for process to run.
389         acquire(&ptable.lock);
390         // reset the variables every raffle
391         golden_ticket = 0;
392         count = 0;
393         total_no_tickets = 0;
394         d++;
395         // calculate tickets total for runnable processes
396
397         total_no_tickets = tickets_total(); //Soma total de bilhetes do sistemas
398         golden_ticket = random_at_most(total_no_tickets); // ticket sorteado randômicamente;
399     }
```

Figura 4. Escalonador por loteria parte 1.

```
383     for(;;){
384
385         // Enable interrupts on this processor.
386         sti();
387
388         // Loop over process table looking for process to run.
389         acquire(&ptable.lock);
390         // reset the variables every raffle
391         golden_ticket = 0;
392         count = 0;
393         total_no_tickets = 0;
394         d++;
395         // calculate tickets total for runnable processes
396
397         total_no_tickets = tickets_total(); //Soma total de bilhetes do sistemas
398         golden_ticket = random_at_most(total_no_tickets); // ticket sorteado randômicamente;
399
400         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
401
402             if(p->state != RUNNABLE){
403                 count += p->tickets; //Soma count mesmo não sendo runnable
404                 continue;
405             }
406
407             if (golden_ticket < count || golden_ticket > (count + p->tickets)){
408                 count += p->tickets; //Soma count mesmo não sendo runnable
409                 continue;
410             }
411         }
```

Figura 5. Escalonador por loteria parte 2.

```

416 //EXIBE INFORMACAO DOS PROCESSOS
417 occurrences[p->pid]++; // Incrementa quantidade de ocorrências por processo
418 if(d % 100 == 0){
419     procdump(occurrences);
420     cprintf("\n");
421 }
422
423 // Mudança de contexto e estado.
424 c->proc = p;
425 switchvm(p);
426 p->state = RUNNING;
427 switch(&(c->scheduler), p->context);
428 switchkvm();
429
430 // Process is done running for now.
431 // It should have changed its p->state before coming back.
432 c->proc = 0;
433 break;
434
435 }
436
437 release(&ptable.lock);
438 }
439 }
440

```

Figura 6. Escalonador por loteria parte 3.

O algoritmo é simples: no laço infinito do escalonador é feito o somatório de bilhetes existentes e sorteado um número aleatoriamente (créditos ao **Siddharth Singh** por criar as bibliotecas *rand.h* e *rand.c* para gerar números aleatórios <https://01siddharth.blogspot.com/>).

Após possuir o bilhete sorteado, o escalonador procura pelo processo em que seu estado seja 'RUNNABLE' e que possua o bilhete sorteado.

A distribuição de bilhetes é concatenada com o processo anterior. Por exemplo: vamos criar três processos, o primeiro com 10 bilhetes, o segundo com 20 bilhetes e o terceiro com 30 bilhetes. O primeiro processo possui os bilhetes de [0:9], o segundo possui os bilhetes de [10:29], e o terceiro processo possui os bilhetes de [30:59], sendo que o sorteio é um número entre 0 e 59. A variável que controla a busca do processo vencedor é a *int count*.

Portanto será feito a preempção da CPU somente para o processo que estiver com o estado em 'RUNNABLE' e possuir o bilhete sorteado.

Implementamos também um vetor de ocorrências de processo para gerar informações relevantes ao uso do escalonador por loteria. Esse vetor tem o tamanho da quantidade de processos que o xv6 pode instanciar, e cada posição do vetor é equivalente ao seu identificador. A posição do vetor é incrementada sempre que o processo é executado. Dessa forma podemos calcular a porcentagem de uso da CPU que o processo utiliza em tempo real.

Para testar o escalonador, no arquivo *lottery.c*, é realizado um laço da quantidade desejada para criação de processos, sendo que cada processo criado no laço é passado os bilhetes em função do índice do laço.

4.3.1. Bug

Durante implementação nos deparamos com um bug: o último processo do laço criado para teste ficava sempre no estado 'ZOMBIE'. Não entendemos o porquê. Dessa forma (tentativa de correção) foi criado um processo após o laço com um único bilhete, assim todos os processos, inclusive esse de um único bilhete, ficam no estado 'RUNNABLE'. Ficamos abertos para estudo e correção desse erro. A Imagem 7 mostra com clareza o arquivo de teste e o local do bug:

```
4 |
5 | #include "types.h"
6 | #include "stat.h"
7 | #include "user.h"
8 |
9 | #define QTD_PROC 10
10 |
11 | void process_test(int tickets);
12 |
13 | int main() {
14 |
15 |     for (int i = 1; i <= QTD_PROC; i++) {
16 |         process_test(i*100);
17 |     }
18 |
19 |     //Bug: Se não criar este processo o último do laço fica sempre como zombie.
20 |     process_test(1); // Para não bugar.
21 |
22 |     exit();
23 | }
24 |
25 |
26 | void process_test(int tickets){
27 |
28 |     int i = 0;
29 |
30 |     if (fork(tickets)) {
31 |
32 |         // LOOP INFINITO INCREMENTANDO VARIÁVEL PARA NÃO OTIMIZAR
33 |         while (1) i++;
34 |     }
35 | }
36 |
37 |
```

Figura 7. lottery.c, arquivo para testar o escalonador.

4.4. Exibindo os resultados

Por fim, modificamos a função *void procdump(void)* para *void procdump(int *occurrences)*, para que esta receba o ponteiro do vetor de ocorrências do escalonador. Nesta função é formatada uma tabela para exibir informações sobre todos os processos do xv6.

Para testar o escalonador basta entrar na pasta 'xv6-public' via terminal, então digitar o comando: 'make; make qemu-nox'. Aguardar o sistema iniciar e então no console do xv6 fazer a chamada de sistema de teste, digitando 'lottery' e pressionando enter. O sistema começa imprimir a tabela com as informações dos processos. A Imagem 8 mostra a tabela formatada com as informações dos processos, sendo o PID o identificador do processo, NAME é o nome da chamada de sistema que foi chamado, STATE é o estado que o processo se encontra, QTD-T é a quantidade de bilhetes que o processo possui, OC é a quantidade de ocorrência que o processo ganhou a CPU, PROC é a porcentagem em tempo real de uso da CPU em cada processo e por fim, ESTI é a porcentagem estimada de CPU que cada processo pode ter. O cálculo das porcentagens é generalizado conforme (1) e (2) respectivamente.

$$PROC[PID] = \frac{OC[PID]}{\sum OC} \cdot 100 \quad (1)$$

$$ESTI[PID] = \frac{QTD_T[PID]}{\sum QTD_T} \cdot 100 \quad (2)$$

PID	NAME	STATE	QTD_T	OC	PROC	ESTI
1	init	sleep	0	17	0%	0%
2	sh	sleep	64	19	0%	1%
3	lottery	runnable	64	427	1%	1%
4	lottery	runnable	100	612	1%	1%
5	lottery	runnable	200	1260	3%	3%
6	lottery	runnable	300	1827	5%	5%
7	lottery	runnable	400	2484	7%	7%
8	lottery	runnable	500	3213	9%	8%
9	lottery	runnable	600	3722	10%	10%
10	lottery	runnable	700	4339	12%	12%
11	lottery	runnable	800	5159	14%	14%
12	lottery	runnable	900	5613	16%	16%
13	lottery	runnable	1000	5852	16%	17%
14	lottery	runnable	1	0	0%	0%

Figura 8. Tabela com as informações dos processos durante o teste.

5. Conclusão

Analisando a Figura 8, percebemos que após um tempo de processo do teste a porcentagem real processada tende a ser igual a porcentagem estimada em cada processo. Isso prova que o escalonador por loteria funciona. Os processos utilizaram a CPU proporcionalmente à quantidade de bilhete que eles possuem, desta forma evita-se o problema de *starvation*, pois mesmo processos que possuam poucos bilhetes serão executados.

Referências

- de Oliveira, R. S., da Silva Carissimi, A., and Sirineo, S. (2010). *Sistemas Operacionais*. Bookman, 4th edition.
- Tenenbaum, A. S. (2011). *Sistemas Operacionais Modernos*. PEARSON, 3rd edition.