Trabalho I - Escalonamento por Loteria / Lottery Scheduler

Eduardo Tonatto, Gabriel H. Moro

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

edtonatto@gmail.com, gabrielhmoro@gmail.com

1. Descrição do planejamento e implementação

Nesta seção do artigo/relatório será explicado como foi planejado e implementado o escalonador, bem como as modificações realizadas em cima do código-fonte original do xv6.

1.1. Planejamento do trabalho

Para a realização do trabalho foi necessário, primeiramente, para a implementação, identificar os arquivos, estruturas e funções em que se havia necessidade de modificação, como ocorreriam estas modificações. Para o artigo houve o planejamento dos tópicos a serem escritos e da forma a serem estruturados.

1.2. Implementação e modificações

O escalonador por loteria, ou em inglês, *lottery scheduler*, funciona a partir da atribuição de bilhetes (*tickets*) aos processos, sendo que cada processo deve receber no mínimo um bilhete até no máximo N bilhetes, e não deve ser permitido um processo receber zero bilhetes ou valores negativos de bilhetes (caso isso aconteça estes processos "morreriam de fome"(*starvation*), pois nunca seriam sorteados pelo escalonador), esta atribuição de bilhetes deve ocorrer assim que os processos iniciais são criados, e a cada chamada de fork realizada pelo sistema.

O funcionamento do escalonador por loteria se dá da seguinte maneira: o escalonador, através de algum algoritmo de randomização, no caso desta implementação utiliza-se o "Gerador congruente linear"(em inglês, *Linear congruential generator*), sorteia um valor aleatório de algum bilhete (apenas bilhetes de processos com o estado PRONTO (*RUNNABLE*) poderão entrar no sorteio), este bilhete é o bilhete vencedor, logo o processo que possuí-lo será o vencedor, e assim terá sua vez na CPU.

Segue abaixo as modificações realizadas, explicitando o nome do arquivo modificado e comentários referentes as alterações realizadas no código do xv6:

Arquivo: proc.h

```
// Definições dos números máximo e padrão para os tickets
2 - #define MAX_TICKETS 101
3 - #define DEFAULT_TICKETS 10

// Dentro da struct proc
56 - int tickets; // Nº de tickets do processo
57 - int chamado; // Nº de vezes que o processo
                     foi selecionado pelo escalonador
```

Arquivo: proc.c

```
// Definição de um controle de DEBUG
9 - #define DEBUG

// Declaração de uma variável global para
// o controle da aleatoriedade do escalonador
11 - unsigned int lcg = 3;




// Dentro da função \void userinit(void)"
144 - p->tickets = DEFAULT_TICKETS; // Processo incial deve receber
                                       número padrão de processos,
                                       caso contrário o scheduler
                                       nunca irá selecioná-lo

// Dentro da função \int fork(void)"
184/5 - int fork(int numTickets) // Agora nossa função fork precisa
                                    receber uma variável \numTickets"
                                    para definir com quantos tickets
                                    o novo processo será criado
196 - np->chamado = 0; // Seta o número de vezes que o processo
                          recém criado foi chamado para zero

// Agora precisamos atribuir o valor
   de \numTickets" ao novo processo \np"
208 - if(numTickets != 0){ // Verifica se foi passado um valor
                              de tickets para o processo
209 -     if(numTickets > MAX_TICKETS){ // Verificar se número de
                                           tickets é maior que o limite
210 -         np->tickets = MAX_TICKETS; // Se for maior que o limite,
                                            atribuimos o limite
                                            para o processo
211 -     }else{// Verifica se o número de tickets é menor que o limite
212 -         np->tickets = numTickets; //Se for menor que o limite,
                                         atribuimos o \numTickets"
                                         para o processo
213 -     }
214 - }else{ // Se número passado for zero:
215 -     np->tickets = DEFAULT_TICKETS; // Atribui-se o valor padrão
                                            ao processo
216 - }


// Debug:
230 - #ifdef DEBUG
231 - cprintf("PROCESSO CRIADO. PID: %d TICKETS: %d\n",np->pid,np->tickets);
232 - #endif
```

```
// Criação de uma função para tratar da aleatoriedade do scheduler
333 - unsigned int lcg_rand(unsigned int state)
334 - {
335 -     state = ((unsigned int)state * 48271u) % 0x7fffffff;
336 -     return state;
337 - }

// Criação de uma função para retornar
   o total de tickets existentes,
   somente dos processos com estado RUNNABLE
339 - int getTotalTickets(void){
340 -   struct proc *p;
341 -   int totalTickets=0;
343 -   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
344 -     if(p->state == RUNNABLE){
345 -       totalTickets += p->tickets;
346 -     }
347 -   }
348 -   return totalTickets;
349 - }

// Dentro da função \void scheduler(void)"
359 - void
360 - scheduler(void)
361 - {
362 -   int totalTickets = 0;
363 -   int escolhido;
365 -   struct proc *p;
366 -   struct cpu *c = mycpu();
367 -   c->proc = 0;

369 -   for(;;){
371 -     // Enable interrupts on this processor.
372 -     sti();

374 -    // Loop over process table looking for process to run.
375 -     acquire(&ptable.lock);

377 -     totalTickets = getTotalTickets(); // Retorna o total de tickets
                                              para verificação posterior

379 -    if(totalTickets > 0){ // Se existem tickets a serem sorteados,
                               entra no if, pelo menos 1 ticket
380 -       escolhido = lcg = lcg_rand(lcg) % totalTickets + 1;
                  // Sorteia um dos tickets e salva este valor em
                     'escolhido', sempre atualizando junto o valor 'lcg'
                      que controla a randomização
381 -       if(totalTickets < escolhido)
```

```
                    // Caso a operação anterior (linha 380) retorne um
                       valor fora do range de tickets, entra no if
382 -         escolhido %= totalTickets; // E aqui coloca o valor dentro
                                          do range
383 -       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                    // Percorre toda tabela de processos
384 -         if(p->state == RUNNABLE) // Se o processo atual é RUNNABLE,
                                       está pronto:
385 -           escolhido -= p->tickets; // Então retira-se o seu valor
                                          de tickets(p->tickets)
                                          escolhido(através do sorteio)
386 -         if(p->state != RUNNABLE || escolhido >= 0)
                    // Se o processo atual não é RUNNABLE,
                       logo este não será o processo escolhido,
                       ou se o valor em \escolhido" for maior
                       que zero também não atingimos o processo
                       com o bilhete sorteado
387 -           continue;
                    // Continua-se até termos um processo RUNNABLE
                       e que tenha tornado o valor de escolhido
                       menor que 0 na operação da linha 385

392 -         c->proc = p;
393 -         switchuvm(p);
394 -         p->state = RUNNING;
395 -         #ifdef DEBUG // Debug para casos de teste
396 -         cprintf("PROCESSO %d ESTÁ COM A CPU AGORA.\n", p->pid);
397 -         #endif
398 -         p->chamado++; // Incrementa a variável 'chamado' para
                              sabermos ao final, quantas vezes cada
                              processo foi selecionado.

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

405 -         c->proc = 0;
406 -         break;
407 -       }
408 -     }
409 -     release(&ptable.lock);

411 -   }
412 - }


// Dentro da função \void procdump(void)":
// Modificação do anterior \cprintf" existente,
   incrementando duas variáveis a serem printadas,
   para fins de teste, as variáveis \p->tickets"
```

```
   e \p->chamado" foram adicionadas para testarmos
583 - cprintf("%d %s %s %d %d", p->pid, state, p->name, p->tickets,
      p->chamado);
```

Arquivo: Makefile

```
177 - _teste\ // Inclusão do arquivo teste.c no make
250 - teste.c\ // Inclusão do arquivo teste.c no make
```

Os próximos arquivos: ”defs.h”, ”forktest.c”, ”init.c”, ”sh.c”, stressfs.c”, ”sysproc.c”, ”user.h”, ”usertests.c”e ”zombie.c”foram todos modificados em chamadas da função fork, que esta contida na declarada na linha 184 do arquivo ”proc.c”. Anteriormente a nossa função fork não esperava por parâmetros de entrada, e agora ela espera por um valor inteiro que define com quantos tickets o processo sera criado, então é necessário modificar em todos os arquivos que anteriormente usavam o fork desta maneira: fork(), e agora passam a ser usados desta maneira: fork(0).

Arquivo: defs.h

```
108 - int fork(int); // Anteriormente \(void)"
```

Arquivo: forktest.c

```
24 - pid = fork(0); //Anteriormente \pid = fork();"
```

Arquivo: init.c

```
24 - pid = fork(0); //Anteriormente \pid = fork();"
```

Arquivo: sh.c

```
186 - pid = fork(0); //Anteriormente \pid = fork();"
```

Arquivo: stressfs.c

```
27 - if(fork(0) > 0); // Anteriormente \if(fork() > 0);"
```

Arquivos: sysproc.c

```
// Dentro da funçao \int sys_fork(void)":
10 - int
11 - sys_fork(void)
12 - { // Alterações para que a chamada fork
          possa receber um inteiro como parametro
13 -   int num;
14 -   argint(0, &num);
15 -   return fork(num);
16 - }
```

Arquivo: user.h

```
5 - int fork(int); // Anteriormente \int fork(void)"
```

Arquivo: usertests.c

```
49 - pid = fork(0); //Anteriormente \pid = fork();"
94 - pid = fork(0); //Anteriormente \pid = fork();"
315 - pid = fork(0); //Anteriormente \pid = fork();"
365 - pid1 = fork(0); //Anteriormente \pid1 = fork();"
370 - pid2 = fork(0); //Anteriormente \pid2 = fork();"
376 - pid3 = fork(0); //Anteriormente \pid3 = fork();"
410 - pid = fork(0); //Anteriormente \pid = fork();"
435 - if((pid = fork(0)) == 0); //Anteriormente
                                  \if((pid = fork()) == 0)"
478 - pid = fork(0); //Anteriormente \pid = fork();"
530 - pid = fork(0); //Anteriormente \pid = fork();"
593 - pid = fork(0); //Anteriormente \pid = fork();"
781 - pid = fork(0); //Anteriormente \pid = fork();"
829 - pid = fork(0); //Anteriormente \pid = fork();"
865 - pid = fork(0); //Anteriormente \pid = fork();"
1387 - pid = fork(0); //Anteriormente \pid = fork();"
1436 - pid = fork(0); //Anteriormente \pid = fork();"
1499 - pid = fork(0); //Anteriormente \pid = fork();"
1519 - if((pids[i] = fork(0)) == 0) // Anteriormente
                                     if((pids[i] = fork()) == 0)
1572 - if((pid = fork(0)) == 0); //Anteriormente
                                   \if((pid = fork()) == 0)"
1618 - pid = fork(0); //Anteriormente \pid = fork();"
1709 - pid = fork(0); //Anteriormente \pid = fork();"
```

Arquivo: zombie.c

```
11 - if(fork(0) > 0) // Anteriormente \if(fork() > 0)
```

2. Testes para avaliação do funcionamento de desenvolvimento do escalonador

Para determinar o funcionamento do escalonador foram realizados testes da seguinte forma: houve a criação de um arquivo de testes, chamado de ”teste.c”, onde foi definida uma constante N representando a quantia de processos a serem criados, há um laço de repetição indo de n até N, onde n inicialmente vale zero, e dentro dessa repetição são chamados os N forks, sempre passando um valor que define a quantia de bilhetes do novo processo que será criado, esse número de bilhetes é controlado pela variável de nome ”lcg”, a variável que recebe o valor retornado pela função ”lcg_rand”que gera um valor aleatórios de bilhetes para o processo. Após gerados os N processos, é utilizado o comando Ctrl + P do QEMU, alterado dentro da função ”void procdump(void)”na linha 583, para mostrar na tela o PID, o estado e o nome do processo, bem como a sua quantia de bilhetes e quantas vezes foi chamado pelo escalonador.

O arquivo ”teste.c”utilizado para a realização dos testes de fork e escalonamento é o seguinte:

```
1 - #include "types.h"
2 - #include "stat.h"
```

```
3 - #include "user.h"
4 - #include "fs.h"

6 - #define N 30 // N é o numero de processos que serao criados

8 - unsigned int lcg = 3; // Variavel de controle da randomização
                            dos tickets

10 - unsigned int lcg_rand(unsigned int state) // Função de randomização
11 - {
12 -     state = ((unsigned int)state * 48271u) % 0x7fffffff;
13 -     return state;
14 - }

17 - void testfork(void){
18 -  int n, pid;

20 -   printf(1, "Executando testes\n");

22 -   for(n = 0; n < N; n++){
23 -     lcg = lcg_rand(lcg) % 101; // Sorteia um valor para definir
                                      a quantia de ticket do processo
                                      a ser criado
24 -     pid = fork(lcg);
25 -     if(pid < 0) break;
26 -     if(pid == 0){
27 -       for(;;);
28 -     }
29 -   }

31 -   if(n == N){
32 -     printf(1, "Fork foi chamado %d vezes!\n", N);
33 -   }

34 -   for(; n > 0; n--){
35 -     wait();
36 -   }

39 -   exit();
40 - }

42 - int main(void){
43 -   testfork();
44 -   exit();
45 - }
```

Foram realizados alguns testes durante a implementação do escalonador, estes são os resultados mais relevantes e as suas implicações no progresso da implementação:

OBS.: Os PIDs das tabelas começam em 4, pois não são considerados os dois

processos de iniciação do xv6 e o processo chamado para criar os processos gerados para o teste.

Teste 1:

PID	BILHETES	CHAMADAS
4	1	2
5	2	44
6	3	53
7	4	50
8	5	62
9	6	69
10	7	67
11	8	75
12	9	68
13	10	73

Neste teste foi implementado um arquivo de teste alternativo, onde criávamos processos em ordem, sempre definindo sua quantia de bilhetes de acordo com a quantia de bilhetes do processo anterior acrescido de um. Com este teste, nota-se, aparentemente, que o algoritmo de escalonamento esta funcionando de maneira a sortear mais vezes processos que possuem mais bilhetes, mas foi possível perceber que o número de vezes que um processo ou outro estavam sendo chamados eram muito próximos um do outro, então foi criado um caso de teste semelhante, porém fornecendo número de bilhetes em ordem decrescente. Com o resultado deste teste com os bilhetes decrescendo percebeu-se que o padrão de vezes que cada processo foi chamado se manteve o mesmo da tabela acima, onde o bilhete com mais bilhetes era o primeiro a ser criado e o com menos bilhetes o último, observou-se que quanto mais novo o processo, mais chamado ele era.

Isso se dava principalmente pelo fato de que, no escalonador, há um laço de repetição que percorre toda tabela de processos (este iniciado na linha 383 do arquivo ”proc.c”) e que no final deste laço, após o escalonador selecionar o próximo processo, ele continuava a selecionar os processos subsequentes ao selecionado ao invés de selecionar um novo processo a partir dos bilhetes, sendo assim, quanto ”mais recentemente criado”o processo, mais vezes ele seria chamado. Isso implicou no acréscimo de um ”break”no final deste laço (linha 406 de ”proc.c”) para que, toda vez ao achar um processo, ele reiniciasse do inicio da tabela de processos, mantendo a aleatoriedade das chamadas

Teste 2:

PID	BILHETES	CHAMADAS
4	80	1109
5	46	637
6	82	1107
7	32	392
8	79	1076
9	53	762
10	33	478
11	72	922
12	1	19
13	94	1076

Para este teste, foi feita uma implementação de uma versão do arquivo ”teste.c”semelhante a versão final, porém gerando apenas 10 processos. Comparado ao Teste 1, nota-se um avanço significativo, pois neste os processos são gerados com bilhetes aleatórios e há a correção do problema citado no teste anterior. Porém, após algumas execuções deste algoritmo de teste observou-se que alguns processos iniciavam com uma quantia de chamadas observavelmente errada, pois a quantia de chamadas de processos recém criados excedia 0 ou qualquer quantia próxima e possível. A resolução deste problema se deu quando foi observado que a variável de chamada, uma variável criada na estrutura proc que se encontra no arquivo proc.h usada para que se tenha o controle da quantia de chamadas de um processo, não era inciada com 0, mas sim com algum valor aleatório, comumente chamado ”lixo de memória”. Então, na linha 196 do arquivo ”proc.c”foi adicionada a seguinte instrução: ”np→chamado = 0”. Com isso, o novo processo (np) tinha seu valor de chamadas definido como 0 logo após sua criação, evitando com que se tivesse um valor incorreto no controlador.

Teste 3:

PID	BILHETES	CHAMADAS
4	1	0
5	100	1294

Neste caso de teste, onde o arquivo de teste cria um processo com o máximo de bilhetes e outro com o mínimo e, com isso, foi encontrado outro erro no escalonador. Sabe-se que o xv6 possui duas CPUs, logo, com a existência de apenas dois processos, independentemente da sua quantia de bilhetes os dois deveriam ser igualmente chamados pelo escalonador, o que visivelmente não ocorria, pois um processo nunca foi chamado. Mas, ao executar este mesmo teste, agora com um processo com o máximo de bilhetes e outro com o valor padrão, percebeu-se que ambos estavam sendo chamados a mesma quantidade de vezes, logo o problema ocorria quando havia apenas um bilhete. Este problema foi resolvido da seguinte forma: ’escolhido = lcg = lcg_rand(lcg) % *totalTickets* + 1;’, acrescentando ao final do comando o operação ”+1”, corrigindo o range do sorteio, que ia de 0 à *totalTickets* - 1, para 1 à *totalTickets*, tendo em vista que a operação de módulo (%) retorna valores que vão de zero até o valor anterior ao do módulo, no caso, *totalTickets* - 1. Outro erro encontrado foi, logo após a linha 192 do arquivo proc.c (dentro do comando fork), havia um ”release(ptable.lock);”, fazendo com que em alguns casos o xv6 roda-se apenas uma CPU, e, tirando este comando, aliado com a solução apresentada anteriormente, notou-se que o funcionamento ficou como o esperado.

Teste 4: Este é o teste final, onde foi utilizado o arquivo ”teste.c”bem como citado anteriormente, criando 30 processos com os seus valores de bilhetes aleatórios, foram registrados os estados desses processos a cada 10 minutos durante um período de meia hora.

10 minutos de execução:

PID	BILHETES	CHAMADAS
4	80	6048
5	46	2590
6	82	4667
7	32	1540
8	79	2556
9	53	5762
10	33	4337
11	72	5217
12	1	2
13	94	5983
14	49	5584
15	61	4517
16	78	5577
17	60	4706
18	85	9651
19	11	1228
20	24	118
21	34	1473
22	65	6526
23	50	3762
24	54	3252
25	26	240
26	20	998
27	62	4912
28	71	7570
29	8	159
30	45	4142
31	89	7277
32	84	7795
33	18	1210

20 minutos de execução:

PID	BILHETES	CHAMADAS
4	80	11959
5	46	4866
6	82	9053
7	32	2822
8	79	4877
9	53	11199
10	33	8604
11	72	10201
12	1	4
13	94	11597
14	49	10859
15	61	8427
16	78	10692
17	60	9245
18	85	203444
19	11	2394
20	24	242
21	34	2787
22	65	12789
23	50	7590
24	54	6473
25	26	529
26	20	1967
27	62	9923
28	71	14990
29	8	234
30	45	8295
31	89	16051
32	84	15450
33	18	2414

30 minutos de execução:

PID	BILHETES	CHAMADAS
4	80	18034
5	46	7181
6	82	13466
7	32	4136
8	79	7242
9	53	16804
10	33	12938
11	72	15168
12	1	8
13	94	17268
14	49	16355
15	61	12427
16	78	15877
17	60	13776
18	85	30066
19	11	3682
20	24	340
21	34	4084
22	65	19103
23	50	11538
24	54	9767
25	26	813
26	20	2932
27	62	15099
28	71	22487
29	8	301
30	45	12496
31	89	23740
32	84	23108
33	18	3616

Após este teste final, nota-se que o sistema de escalonamento por loteria está funcional, pois é visível que os processos com mais bilhetes possuem maior prioridade perante o escalonador. Vale salientar que para uma melhor visualização do total funcionamento do escalonador é preciso que o escalonador fizesse muitos escalonamentos. Neste caso de teste, que foi executado por meia hora alguns processos. Embora alguns processos possuíssem menos bilhetes que outros, foram sorteados mais vezes. Isso pode ocorrer devido à ”idade”do processo, pois processos mais antigos já teriam ”sido chamados”alguma quantia de vezes e por isso os mais recentes ”começam atrás”na quantidade de chamadas, ou pelo fato de que, como o sorteio é aleatório e a diferença de bilhetes entre eles não é tão grande, pode ocorrer que processo com menos bilhetes acabem ganhando dos que possuem mais, ou seja, se deixássemos esse algoritmo ”teste.c”rodando por algumas horas, provavelmente haveria uma distribuição melhor em relação à chamadas/bilhete, fazendo com que a probabilidade de cada processo vencer o sorteio de acordo com o total

de bilhetes e a quantia de bilhetes dele se tornasse mais parecida com o resultado final da execução do algoritmo.

3. Conclusões

A partir da implementação do escalonador e dos testes realizados, chega-se a conclusão de que o escalonador por loteria passa a ter uma distribuição ”correta”, ou seja, uma distribuição que corresponde a prioridade, tendo como a quantia de bilhetes, somente após muitas interações, o que pode não ser, de fato, eficiente, devido a demora para que o escalonamento comece a ter uma distribuição justa, além de que, por se tratar de um algoritmo que seleciona aleatoriamente, apesar de ser extremamente baixa, existe a possibilidade de alguns processos nunca serem selecionados (ou podem demorar muito a serem chamados) ou alguns outros problemas resultantes desta aleatoriedade.