UNIVERSIDADE FEDERAL DA FRONTEIRA SUL - UFFS

ESCALONADOR DE PROCESSOS POR LOTERIA (LOTTERY SCHEDULER)

Desenvolvido pelos acadêmicos do curso de Ciência da Computação,

Jardel O. Duarte e Jod F. Pierre.

Chapecó-SC

2019

1 - Introdução

No projeto proposto re-implementar o escalonador de processos do xv6 definindo-o de forma a buscar tickets vencedores exatamente como de uma loteria convencional, assim que o sorteio termina o processo do bilhete(tickets) sorteado ganha a CPU. Foram criadas funções e feito alterações nos arquivos do xv6 para composição deste trabalho.

1.1 - Xv6

“[...] No outono de 2002, foi criado para ensinar engenharia de sistemas operacionais. Nas palestras do curso, a turma trabalhou no Sixth Edition Unix(também conhecido como V6) usando o famoso comentário de John Lions. Nas tarefas de laboratório, os alunos escreveram a maior parte de um sistema operacional exokernel, eventualmente chamado Jos, para o Intel x86. A exposição dos alunos a vários sistemas - V6 e Jos - ajudou a desenvolver um senso do espectro de projetos de sistemas operacionais. V6 apresentou desafios pedagógicos desde o início. Os alunos duvidaram da relevância de um sistema operacional obsoleto de 30 anos de idade, escrito em uma linguagem de programação obsoleta (pre-K & R C) em execução em um hardware obsoleto (o PDP-11). Os alunos também se esforçaram para aprender os detalhes de baixo nível de duas arquiteturas diferentes (o PDP-11 e o Intel x86) ao mesmo tempo. No verão de 2006, decidimos substituir o V6 por um novo sistema operacional, xv6, modelado no V6, mas escrito em ANSI C e rodando em máquinas Intel x86 com multiprocessador. O uso do x86 pelo xv6 o torna mais relevante para a experiência dos alunos do que a V6 e unifica o curso em torno de uma única arquitetura. A adição de suporte a multiprocessadores exige o tratamento simultâneo de bloqueios e threads (em vez de usar soluções de casos especiais para uniprocessadores, como ativar/desativar interrupções) e ajuda a relevância. Finalmente, escrever um novo sistema nos permitiu escrever versões mais limpas das partes mais difíceis da V6, como o agendador e o sistema de arquivos. 6.828 substituiu xv6 por V6 no outono de 2006.” (MIT-pdos, 2012).

1.2 - Curiosidades(features) Kernel UNIX

- O kernel do Linux possui um gerador de número aleatório verdadeiro (TRNG)

incorporado que é utilizado para gerar bilhetes de loteria vencedores.

- O kernel Unix não fornece suporte p/ matemática de valores ponto flutuante, observações onde o resultado de divisão inteira em um bloco de código provou essa problemática. Ocorrendo possibilidades de perda total dos bilhetes acumulados de base efetivos devido a esse truncamento.
  “Quando o número da loteria vencedora estiver perto ou no topo da faixa de bilhetes de loteria, ou seja, quase igual ao número total de bilhetes efetivos de base; o sorteio da loteria poderia resultar em nenhum vencedor se muitos os bilhetes foram perdidos devido ao truncamento”(ZEPP, 2012).
- “Idle_task” é uma função que é executada em casos de nenhum bilhete ser sorteado, retornando uma fila(runqueue), esse seria o default do Unix Kernel por padrão, em caso de nenhum bilhete ser selecionado essa fila de execução é chamada e outro processo recebe a CPU em estado de espera ou pausado.
- Como visto na tese Calendário de Loteria no Kernel Linux: um olhar mais próximo de Zepp, existe a possibilidade de aplicar o algoritmo do escalonador de processos de loteria em processamento de servidores como o Apache, os exemplos de Zepp foram com protocolos httperf.

2 - Metodologia

2.1 - Descrição

Descrição do Trabalho I: Implementar o escalonador de processos baseado em loteria (lottery scheduling). Na instanciação de um processo, deve-se passar 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. Assumir também um número máximo de bilhetes que um processo pode receber.

O que entregar (T1 e T2): Código completo, incluindo códigos dos testes realizados e instruções para execução dos mesmos (em um arquivo nomeado LEIAME). Para reduzir o tamanho do arquivo comprimido, exclua os arquivos objetos antes de comprimir (i.e., execute “make clean” para fazer a limpeza antes de comprimir).

2.2 - Especificação do sorteio aleatória

O escalonador de processos por loteria é um algoritmo probabilístico que faz o agendamento para os processos de um sistema operacional. Cada processo recebe

um valor como quantidade de tickets de loteria, e o escalonador sorteia um bilhete aleatório para selecionar o próximo processo a receber a CPU. Esta distribuição não precisa ser uniforme; a única prioridade possível é conceder a um processo mais tickets, desta forma relativamente este processo terá maior possibilidade em ser sorteado. Este é apenas um dos modelos dos possíveis a serem implementados, existe os algoritmos de escalonamento por menor trabalho a seguir e escalonamento de compartilhamento justo.

2.3 - Características

- Os processos devem receber no mínimo 1 bilhete;
- Os processos são limitados a uma entrada de tamanho N_MAX de bilhetes;
- Caso não forem atribuídos bilhetes ocorre starvation e o processo nunca ganhará a CPU;
- A atribuição dos bilhetes devem ser iniciado na criação dos processos, em cada chamada do fork do sistema;

2.4 - Implementação da lógica de loteria e alterações no xv6

Disponivel em: https://github.com/jodfedlet/OperatingSystem/tree/master/loteria e também neste repositório.

Alterações no xv6:

- proc.h

```
41  struct proc {
42    int tickets; // Quantidade de bilhetes do processo
43    int callback;//Quantidade de vezes que o processo foi selecionado pelo Escalonador
```

  Necessário a criação de variáveis, tickets para receber a entrada, e um callback para devolver a quantidade de vezes que o processo foi vencedor do sorteio.

- proc.c

```
10  #define MAX 100 /*definindo 100 como total MAX*/
11  #define DEFAULT 10 /*definindo default 10*/
12  unsigned int lcg = 3; /*controle de aleatoriedade do escalonador*/
```

  Definimos variáveis no escopo.

```
74  static struct proc* allocproc(int number_tkts){ //nesta função adicionamos
75    //um parametro com o numero de tickets

91    //verifica os numeros tickets
92    if(number_tkts > MAX) p->tickets = MAX;
93    else if(number_tkts < 1) p->tickets = DEFAULT;
94    else p->tickets = number_tkts;
95    p->callback = 0; //zeramos o contador de bilhetes vencedores
96    //O bilhete recebe somente se o sorteado estiver no invervalo definido
```

Ainda na função proc criamos uma verificação que determina os limites de cada processos garantindo a quantidade mínima de tickets e zeramos o contador dos bilhetes sorteados.

```
129    p = allocproc(DEFAULT); //alocando e passando o default por garantia
```

Na função void userinit passamos default, sabendo que trocamos a função allocproc que recebe um parâmetro.

```
188    if((np = allocproc(number_tkts)) == 0){ //se numero de tickets igual a 0
189      return -1; //encerra o processo
190    }
```

Garantimos que o número de n processos é diferente de 0 para podermos. Se sim encerra.

```
313  //Função para retornar a escolha aleatória do processo
314  int lcg_lotttery_random(int state){
315     return ((unsigned int)state * 48271u) % 0x7fffffff;
316  }
```

Foi criada uma função que gera e retorna de um número aleatório.

```
339  void scheduler(void){
340
341    int sum_tkts_scheduler = 0; /*variavel contador*/
342    int sorted_ticket; /*variavel recebe o bilhete sorteado*/
```

Nesta função foi onde ocorreram as maiores alterações, no escopo definimos 2 variáveis, soma de bilhetes (sum_tkts_scheduler e sorted_ticket).

```
354       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ // percorre a tabela de processo
355         if(p->state != RUNNABLE) continue;
356         sum_tkts_scheduler += p->tickets; //somador recebe os tickets executando
357       }
```

Percorre a tabela de processos e verifica se estado é diferente de executando e contínua, do contrário incrementa a variável sum_tkts_scheduler o bilhete do processo.

```
359       if(sum_tkts_scheduler > 0){ //verificando se existem processos que contem bilhetes
360         sorted_ticket = lcg_lotttery_random(lcg) % sum_tkts_scheduler+1;
361         //gera um numero aleatório passando o numero de processos

363         if(sorted_ticket < 0) sorted_ticket *= -1; //se bilhete sorteado for negativo
364         //garantimos o multiplicando por -1
365         sum_tkts_scheduler = (sorted_ticket%sum_tkts_scheduler)+1;
366         //garantimos também que esteja no intervalo determinado

368         for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
369             /*dentro do laço que percorre todos os processos*/
370           if(p->state == RUNNABLE){
371             sum_tkts_scheduler -= p->tickets;//escalonador decrementa
372             //o total do bilhete vencedor
373             if(sum_tkts_scheduler <= 0) {//verifica se a soma total é 0 ou negativa
374               p->callback++; // e encrementa o numero de vezes que ele foi selecionado

388               break;
389             }
390           }
391         }
392       }
```

A lógica do escalonamento está toda nesta função, comentados em todas as partes do código garantindo um melhor entendimento.

```
583       /*incluimos 2 variaveis nesse print, afim de analisar os testes*/
584       cprintf("%d %s %s %d %d", p->pid, state, p->name, p->tickets, p->callback);
```

Alteração na função void procdump(void), incluindo p->tickets e p->callback para imprimir junto com o pid e estado e nome do processo.

- Makefile

```
181           _testing\
```

Alterações para gerar os executáveis.

- defs.h

```
108 int             fork(int);
```

- forktest.c

```
24      pid = fork(0);
```

- init.c

```
24      pid = fork(0);
```

- sh.c

```
186    pid = fork(0);
```

- stressfs.c

```
27      if(fork(0) > 0)
```

- sysproc.c

```
10  int
11  sys_fork(void)
12  {
13    int num;
14    if(argint(0, &num)) return(-1);
15    return fork(num);
16  }
```

  Garantindo que a função que anteriormente não estava recebendo parametro agora passe a receber.

- zombie.c

```
11    if(fork(0) > 0)/*alterando para possibilitar a execução do sorteio*/
```

- user.h

```
5   int fork(int);
```

- usertests.c

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

Todos os arquivos acima foram alterados devido a alteração no arquivo proc.c, definimos um parâmetro int fork(int number_tickets) dentro da função que anteriormente era int fork(void), após alguns testes passando a constante 0 alteramos e passamos a testar com o fork(1).

3 - Resultados

Foram implementados testes para verificação do código, a seguir será descrito o teste e posteriormente traremos o relatório de execuções.

3.1 - Código

- test.c

```
1   #include "types.h"
2   #include "stat.h"
3   #include "user.h"
4   #include "fs.h"
5   #define number_process 10 /*definindo a quantidade de processos*/
```

```c
unsigned int lcg = 3; /*variável do controle de randomização*/
unsigned int lcg_lotttery_random(unsigned int state){ /*função para gerar um número
 aleatório que tem um unsigned int como parametro*/
  return ((unsigned int)state *48271u) % 0x7fffffff; /*gera um número aleatório*/
  /*retorna*/
}

void test_fork(void){
        int i, pid;
        printf(1, "Executando os testes\n");
        for(i = 0; i < number_process; i++){ /*laço até quantidade de processos

                lcg = lcg_lotttery_random(lcg) % 100; /*sorteia um bilhete*/

                pid = fork(lcg); /*gera pid da thread como id*/
                if(pid < 0) break;
                else if(pid == 0) for(;;);
        }

        if (i == number_process){
                printf(1, "Fork foi chamado %d vezes.\n", number_process);


int main(int argc, char const *argv[]){
        test_fork();
        exit();
}
```

Primeiro modelo com 10 processos definidos no escopo do código.

3.2 Testes

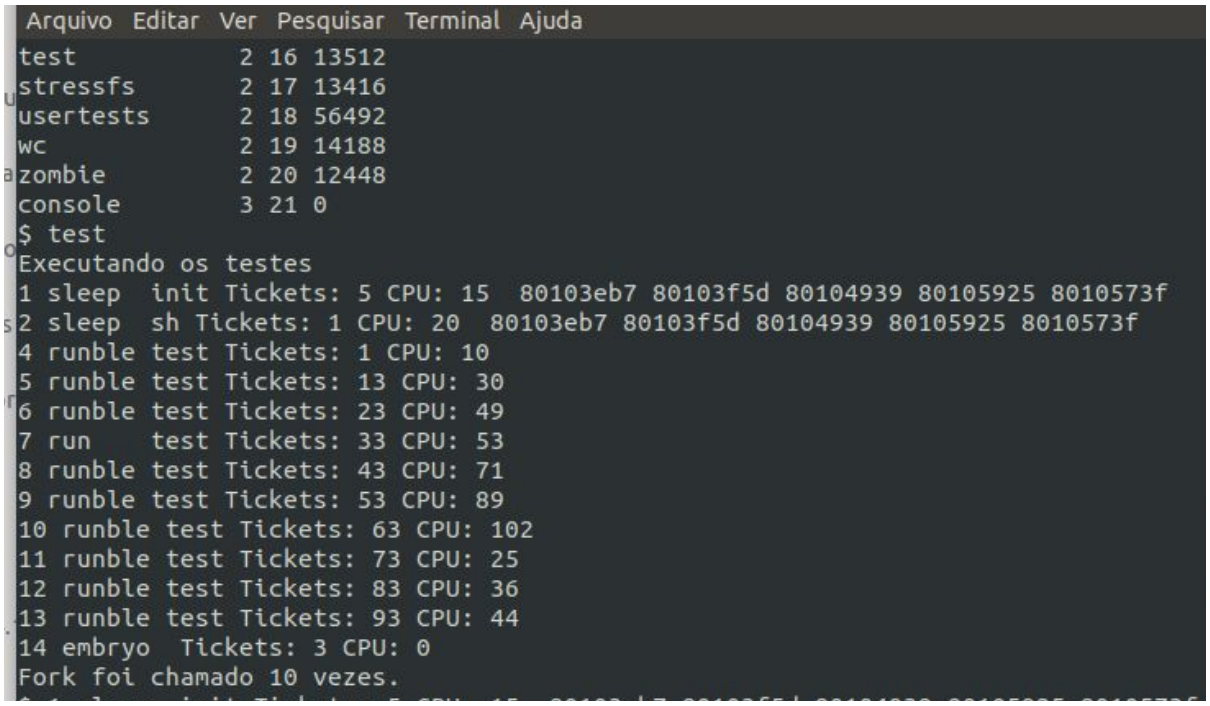

Figura 1 - Terminal Virtual Machine Qemu-kvm

No exemplo acima 4 processos foram iniciados sem concorrer ao sorteio(observe na linha 1 da tabela o sorteio começa no processo de número 4 e vai até o 14, modelo com define 10 processos concorrendo a cpu), provavelmente devido a estarem executando com algum privilégio ou conterem requisitos necessários para ferramenta, em seguida o fork criou 10 processos onde o bilhete 1 venceu 10 vezes na primeira linha, na segunda linha o bilhete 13 venceu 30 vezes. Enfim, posteriormente serão feito maiores testes e com prazos maiores de tempo, possibilitando uma análise mais aprofundada sobre, como por exemplo os possíveis privilégios em registradores tendenciosos a executar algum processo por já ter o endereçamento no buffer, processos de maiores bilhetes com números maiores de vitórias, entre outros.

4 - Execução

- Necessário obter a máquina virtual qemu-kvm disponivel em https://www.qemu.org/
- Fazer download ou clonar este repositório
- No terminal GNU-linux faça:

  :~/user/route$ make

:~/user/route$ make qemu-nox ou :~/user/route$ make qemu

:~/user/route$ testing

E por último pressione (Ctrl + P) para listar os testes.

5 - Conclusão

Após a implementação e análise dos códigos citados, notamos a eficácia na dinamicidade do escalonador por loteria, permitindo alterações nos processos vinculados a CPU assim como as taxas de execução de forma a permitir abstração de bilhetes de loteria. Sabendo que as alocações dos tickets garantem acessos ao processador em proporcional a distribuição dos bilhetes, é garantido aos processos a mesma probabilidade de recursos, e embora os processos com maiores bilhetes possam ser sorteados com maior frequência devido a lógica simples de possibilidades, o método de loteria é uma forma justa de prioridades comparado a outros modelos de escalonamentos, outra observação nestes projeto é a função default, caso nenhum valor de tickets seja iniciado, automaticamente o default supre a necessidade de entrada de tickets de um processo, evitando um futuro starvation.

5 - Referências

COX, Russ. KAASHOEK, Frans. MORRIS, Robert. **Engenharia de Sistemas operacionais do MIT**: MIT-pdos. 2006.

Disponível em: <https://pdos.csail.mit.edu/6.828/2012/xv6.html|> Acesso em 1 de outubro de 2019.

Github disponível em: <git://github.com/mit-pdos/xv6-public.git>

TANENBAUM, S. Andrew. **Modern Operating Systems**. Pearson. 3.ed. 2008.

SINGH, Siddharth. **Implementing lottery scheduling on XV6**. Abril, 2018.

Disponível em: <https://01siddharth.blogspot.com/2018/04/implementing-lottery-scheduling-on-xv6.html> Acesso em 2 de outubro de 2019.

Github disponível em: <https://github.com/siddharthsingh/OS/tree/master/XV6>

ZEPP, David. **LOTTERY SCHEDULING IN THE LINUX KERNEL**: A CLOSER LOOK. 2012. 98p. Thesis the Requirements for the Degree Master of Science in Computer Science presented to the Faculty of California Polytechnic State University, San Luis Obispo, California, 2012.

Disponível em: <https://pdfs.semanticscholar.org/f272/36e8b34042a8ef899832541fb681c01a2ebc.pdf> Acesso em 5 de outubro de 2019.

NEGRÃO, D. Matheus. RIBAS, K. Nikolas. **Escalonador por Loteria - xv6:** Instruções de Execução. Chapecó. 2019. Modelo desenvolvido como requisito da disciplina de Sistemas Operacionais, Universidade Federal da Fronteira Sul, Santa Catarina, Chapecó. 2019.

Disponivel em: <https://github.com/MaNegrao/xv6-Loterry-Scheduling> Acesso em 2 de outubro de 2019.

TONATTO, Eduardo. MORO, H. Gabriel. **Escalonador por Loteria/ Lottery Scheduler**. Chapecó. 2018. Modelo desenvolvido como requisito da disciplina de Sistemas Operacionais, Universidade Federal da Fronteira Sul, Santa Catarina, Chapecó. 2018.

Disponível em: <https://github.com/EdTonatto/UFFS-2018.1-Sistemas_Operacionais> Acesso em 5 de outubro de 2019.