Implementação do escalonador de processos baseado em loteria (lottery scheduling) no xv6

Cleiton de Lima Pinto[1], **Eduardo Mendes Senger**[1], **Patric Venturini**[1]

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

Abstract. *This meta-paper aims to show the fullfiled implementation of the lottery scheduling on xv6. In it will be discussed how to create a process and how / when it will gain CPU processing, considering the implemented lottery algorithm.The meta-paper also contains a step-by-step of what have been done in the implementation and where occured the main changes on xv6 code.*

Resumo. *Este artigo tem como objetivo mostrar a implementação realizada do escalonador baseado em loteria (lottery scheduling) no xv6. Nele será abordado como é feita a criação de um processo e como/quando ele ganhará processamento da CPU, levando em consideração o algorítimo de loteria implementado. O artigo incluirá passo a passo do que foi feito na implementação e onde ocorreram as principais mudanças no código do xv6.*

1. Introdução

Esse projeto nos mostrou como funcionam os escalonadores de processos dentro de um sistema operacional (SO). Nós vimos a maneira em que os processos são escolhidos antes de serem executados e o que o SO faz com eles após seu término de execução.
Cada programa em um sistema operacional é um processo. Para que possam ser executados, eles tem que passar pelo escalonador de processos, já que o número de processos executados ao mesmo tempo é limitado. Um processador de um núcleo pode apenas executar um processo por vez, já em um de dois núcleos é possível rodar dois desses processos simultaneamente.
Nosso objetivo é implementar um escalonador de loteria no sistema operacional *xv6*, que é um SO para fins didáticos. Para a elaboração do projeto nós usamos o código fonte disponibilizado pelos desenvolvedores do *xv6* e nele alteramos as partes necessárias para o funcionamento desse tipo de escalonador.

2. O que é um sistema operacional

Um dos principais objetivos do sistema operacional é gerenciar processos de aplicações afim de abstrair a programação de hardware baixo nível para que programadores possam desenvolver aplicativos sem preocupação de gerenciar o funcionamento de um determinando hardware, podendo assim, ser executada com sucesso. Além disso, o sistema operacional oferece uma interface amigável para o usuário, de tal forma, que ele consiga se comunicar e fazer operações complexas.
Na maioria dos sistemas operacionais, as tarefas são gerenciadas em dois níveis de usuário: kernel (núcleo) e user (usuário). No modo kernel, são implementados mecanismos de proteção e segurança, garantindo maior integridade ao sistema. Já no modo usuário, ele não tem acesso a determinadas operações. Em muitos casos, o modo usuário precisa fazer uma chamada de sistema, para que o modo kernel resolva a situação.

3. O sistema xv6

O xv6 é um sistema operacional criado no MIT para fins didáticos. É uma re-implementação do sistema unix e possui as principais chamadas de sistemas. Ele é implementado para um multiprocessador baseado em x86 moderna usando ANSI C.

4. Processos no xv6

Processo é uma abstração do programa em execução. Mantêm a capacidade de operações de forma que parece ser concorrentes, mesmo quando há apenas uma CPU disponível. A CPU executa processos de programa para programa, podendo ocorrer de dezenas ou centenas de milissegundos. A cada instante a CPU executa somente um programa, no decorrer de um segundo ela pode trabalhar sobre vários programas, dando aos usuários a ilusão de paralelismo.
No xv6, o processo é definido pela seguinte estrutura:

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this proce
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

- *sz* – representa o tamanho em bytes do processo em memória.
- *pgdir*pgdir – indica o endereço para a tabela de página.
- *procstate state* – informa qual o estado do processo, ele pode ser UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE
- *parent* – referência o processo pai.
- *context* – utilizado para armazenar o estado do processo, para que depois ele continue a executar de onde parou.

5. Escalonador

Quando um computador é multiprogramado, ele muitas vezes tem variados processos que competem pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais processos estão simultaneamente no estado de pronto (RUNNABLE); A parte do SO que faz a escolha de qual processo é justamente a do escalonador, e o algoritmo que é usado é chamado de algoritmo de escalonamento.

No xv6, o escalonador possui a seguinte implementação:

```
void
scheduler(void)
{
  struct proc *p;

  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.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

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

  }
}
```

A função *scheduler* é chamada somente uma vez ao iniciar o sistema. Ela funciona em cima um laço infinito, toda interação feita é executado a função *acquire* e *release*. Tudo que for executado entre essas duas funções, fica livre de impasse.

No escalonador é percorrido todos os processos que estão na pilha, por padrão o essa pilha armazena até 64 processos (NPROC).
O escalonador vai escolher o primeiro processo disponível (RUNNABLE) para realizar a troca de CPU. Depois é passado o estado do processo para RUNNING.
O processo só vai voltar a ser chamado novamente, quando estado dele for RUNNABLE novamente.

6. Implementação

O primeiro passo que tomamos foi o de alterar a estrutura dos processos para aceitar os tickets:

```
struct proc {
  .
  .
  .

  int tickets;
};
```

Depois, alteramos a função fork para receber um inteiro e atribui-lo aos tickets, somando uma variável global com total de tickets para a hora do sorteio, assim como a userinit, que cria o primeiro processo, cuidando para dar lock no uso da variável global, evitando comportamentos indesejados:

```
void
userinit(void)
{
  .
  .
  .
  p->tickets = 10;
  acquire(&TotalTickets.lock);
  TotalTickets.totTickets = TotalTickets.totTickets + 10;
  release(&TotalTickets.lock);
}

int
fork(int tickets)
{
  .
  .
  .
  np->tickets = tickets;
  acquire(&TotalTickets.lock);
  TotalTickets.totTickets = TotalTickets.totTickets + tickets;
  release(&TotalTickets.lock);
  *np->tf = *proc->tf;
  acquire(&TotalTickets.lock);
  cprintf("\n— This is the process id: %d, this is the tickets quantity
  release(&TotalTickets.lock);
  .
  .
  .
}
```

E, como vários processos podem alterar uma variável global, foi necessário criar um lock para essa variável de forma que não seja permitido que dois processos acessem a mesma região crítica. Em todo o lugar que essa variável foi utilizada foi necessário adicionar as funções acquire e release, para que funcionasse corretamente caso fosse se-

lecionado para executar o xv6 com mais de uma CPU.

```
struct {

  struct spinlock lock;
  int totTickets;

} TotalTickets;
```

Abaixo um exemplo de uso da variável TotalTickets com o lock:

```
userinit(void)
{
  .
  .
  .
  acquire(&TotalTickets.lock);
  TotalTickets.totTickets = TotalTickets.totTickets + 10;
  release(&TotalTickets.lock);
}
```

Na sequência, alteramos o escalonador para o modo de loteria e criamos uma função ”rand” para o sorteio dos tickets. Então, a cada iteração do escalonador, sorteamos um valor e vamos somando a quantidade de tickets acumulados dos processos até achar o primeiro processo que pode ser executado e tenha o acumulado de tickets maior ou igual que o valor sorteado para ser mandado para o processador:

```
unsigned long next=1;
int rand_int(int rand_max){
    next = next * 1103515245 + 12345;
    int rand=((unsigned)(next/65536) % 32768);
    //above are the default implemenation of random generator with rando
    //need to map it to the
    int result =rand % rand_max+1;
    return result;
}

void
scheduler(void)
{
  struct proc *p;
  int amountOfTickets, luckyTicket;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
```

```
    acquire(&ptable.lock);
    amountOfTickets = 0;
    acquire(&TotalTickets.lock);
    luckyTicket = rand_int(TotalTickets.totTickets);
    release(&TotalTickets.lock);

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

      if(amountOfTickets >= luckyTicket) {
        // Switch to chosen process.
It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.


        proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&cpu->scheduler, proc->context);
        switchkvm();

        // Process is done running for now.
        // It should have changed its p->state before coming back.


        proc = 0;
              break;
      }
    }
    release(&ptable.lock);
  }
}
```

Com isso pronto, alteramos então a forma como era chamado o processo fork nos diversos lugares do programa, passando um número fixo de tickets (10), e também o *sysfork* para aceitar a quantidade de tickets:

```
int
sys_fork(void)
{
  int tickets;
  if(argint(0, &tickets) < 0)
    return -1;
  return fork(tickets);
}
```

Por fim, só adicionamos nas funções kill e exit a subtração dos tickets dos processos que estavam saindo do total de tickets atuais:

```
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    .
    .
    .
    acquire(&TotalTickets.lock);
      TotalTickets.totTickets = TotalTickets.totTickets - p->tickets;
      release(&TotalTickets.lock);
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}

void
exit(void)
{
  .
  .
  .
  acquire(&ptable.lock);
  acquire(&TotalTickets.lock);
  TotalTickets.totTickets = TotalTickets.totTickets - proc->tickets;
  release(&TotalTickets.lock);
  .
  .
  .
}
```

7. Testes

Para testar o funcionamento do nosso código nós utilizamos os seguintes programas testes:
Teste 1:

```
  for (n = 1; n < N; n++) {
    pid = fork(n);
    if(pid < 0)
```

```
      break;
    if(pid == 0) {
      sleep(50);
      printf(1, "\n Proccess %d ended with %d tickets", n, n);
      exit();
    }
  }
```

O teste acima nos permite ver a criação dos forks e a distribuição dos tickets.

Teste 2:

```
  pid = fork(1);

  if(pid == 0) {

    printf(1, "\n Proccess %d STARTED\n", 1);
    for (j = 0; j< BigNum; j++){
      for (z = 0; z< BigNum; z++){
        for (x = 0; x< BigNum; x++){
          printf(1, ""); // this is for the optmization system
        }
      }
    }
    printf(1, "\n Proccess %d ended with %d tickets\n", 1, 1);
    exit();
  }

  pid = fork(10);

  if(pid == 0) {

    printf(1, "\n Proccess %d STARTED\n", 2);
    for (j = 0; j< BigNum; j++){
      for (z = 0; z< BigNum; z++){
        for (x = 0; x< BigNum; x++){
          printf(1, ""); // this is for the optmization system
        }
      }
    }
    printf(1, "\n Proccess %d ended with %d tickets\n", 2, 10);
    exit();
  }
```

O Teste 2 é a criação de várias instancias desse código acima em sequência, buscando um número diferente de tickets para cada processo para ser possível a visualização da distribuição dos processos no escalonador. Com os prints nos lugares certos é possível visualizar a troca de processos em execução e como funciona a troca e a restauração do

contexto de um processo.

8. Conclusões

Concluímos que os processos no sistema operacional xv6 original (antes da nossa alteração) são escalonados de uma forma que não é muito eficiente, pois esses processos são executados a partir de uma tabela, um por um na ordem em que entraram. Vimos que é possível aumentar a utilidade do sistema mudando para um escalonador com mais recursos, como prioridade. Foi possível observar também a funcionalidade das regiões criticas e o que acontece quando mais de uma CPU acessa essas áreas, quando não estão protegidas por um lock, simultaneamente. Também visualizamos como funciona a troca de processos no escalonador quando o processo é preempetível, vendo a troca de contexto quando um processo é escalonado e a restauração desse contexto quando o processo volta a executar.

9. References

Tanenbaum, A. S. (2009). Modern Operating Systems. Pearson Prentice Hall, 3rd edition.

Cox, Russ. Kaashoek, Frans. Morris, Robert (2012). xv6 - a simple, Unix-like teaching operating system