Escalonador de processos *Stride Scheduling*

André Luiz Hofer[1], **Cleiton A. Ambrosini**[1], **Emerson Martins**[1]

[1]Universidade Federal da Fronteira Sul (UFFS)
Curso de Ciência da Computação - Chapecó - Santa Catarina - Brasil

{andreluizhofer,cleito.am}@gmail.com, emer-martins@hotmail.com

Abstract. *XV6 is a simple open source operating system primarily developed by MIT academics for learning purposes. Your base process scheduler is a conventional Round-robin. The present work has with object to analyze and describe how the scheduler base in xv6 works and, then, present a proposal implementation with a working explanation of the stride scheduling.*

Resumo. *O XV6 é um sistema operacional simples de código aberto desenvolvido primeiramente por acadêmicos do MIT para fins didáticos. Seu escalonador de processos base é um Round-robin convencional. O trabalho presente tem como finalidade analisar e descrever como funciona o escalonador base do XV6 e, então, apresentar uma proposta de implementação juntamente com uma explicação de funcionamento do escalonador de processos por passos largos (stride scheduling).*

1. Introdução ao XV6

O XV6 foi desenvolvido no MIT em 2006 por acadêmicos do curso de Engenharia de Sistemas Operacionais. Para o auxílio do entendimento e do funcionamento do sistema o MIT desenvolveu também, um livro explicando a estrutura e o funcionamento do sistema operacional. A distribuição é simples e pequena tal qual foi uma reimplementação da sexta edição Unix.

Por ser uma uma distribuição simples e limitada de funcionalidades o XV6 é tido como uma ferramenta de aprendizado. Embora o sistema operacional carece de funcionalidades presentes em outras distribuições Linux maiores, grande parte dos conceitos e estruturas unix permanecem os mesmos. Por ser um sistema pequeno e simples o XV6 é muito leve e rápido em suas funções levando apenas alguns segundos para compilar, além do mais o XV6 permite a depuração remota.

2. Gerenciamento de Processos no XV6

Um processo, no XV6, consiste em espaço de memória do usuário, que define os segmentos de instruções, dados e pilha, e espaço do kernel (núcleo). O XV6 garante que nenhum processo acesse um espaço de memória que não seja seu, pois o kernel associa um identificador Pid, que é único para cada processo. O kernel mantém atualizado o estado de excução em que o processo se encontra (SLEEPING, RUNNING, RUNNABLE, UNUSED, ZOMBIE OU EMBRYO).

Quando um recurso privado do kernel é solicitado por um processo a partir de uma chamada de sistema, o contexto de execução é desviado ao kernel que executa o pedido e após retorna o segmento do processo. Um processo em estado RUNNABLE é selecionado

pelo escalonador para entrar em execução assim que a CPU estiver disponível. Quando um processo perde a CPU, o estado e os dados deste processo são salvos em registradores para que, quando voltar a ser executado possa continuar a execução de onde parou. Caso ocorra uma interrupção de relógio enquanto um processo estiver em execução, a função yeld() chama o escalonador para seleção de outro processo pronto para executar e o estado do processo que perdeu a CPU muda para RUNNABLE. Quando a interrupção é por um recurso com operação I/O, mas o recurso não está possível, o processo perde a CPU e tem seu estado alterado para SLEEPING.

Os processos no XV6 possuem a seguinte estrutura:

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom kernel stack for process
  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)
};
```

3. Criação de processos no XV6

O XV6 gerencia os processos de forma hierárquica, chamada grupo de processos. Esta hierarquia de processos acontece da seguinte forma: a criação de um processo é feita através da chamada de sistema fork(), o processo criado é chamado processo filho, pois é uma cópia do processo que invocou a chamada fork. O processo filho, também, herda do processo pai o mesmo conteúdo de memória, contudo cada processo tem seu espaço de endereçamento distinto. O XV6 aloca memória do espaço do usuário implicitamente. A chamada fork() aloca memória suficiente para o processo filho copiar o processo pai, a chamada exec() aloca memória necessária para armazenar o arquivo executável. Caso um processo necessite de mais memória, em tempo de execução, poderá chamar sbrk(n), aumentando sua memória em n bytes, o retorno da chamada sbrk(n) será o endereço de memória alocada. A chamada exec() espera um arquivo no formato ELF contendo as instruções e os dados e altera o espaço de memória do processo chamador para uma imagem de memória contendo o arquivo de instruções. Se o arquivo de instruções não gerar erros, as instruções serão executadas pela chamada exec(), partindo pela instrução declarada no cabeçalho do arquivo. A chamada de sistema exit() finaliza a execução de um processo e libera os recursos que o processo adquiriu, como memória e arquivos abertos. A chamada wait() retorna o PID dos processos filhos que finalizaram, se o retorno for -1, significa que o processo pai pode finalizar a execução. Enquanto os processos filhos não finalizarem, a chamada wait() faz com que o processo pai espere por isso. Se um processo pai é morto (killed = 1) enquanto seus processos filhos estavam em execução, os

processos filhos se tornam processos zumbis (i.e. estado é ZOMBIE). A chamada wait() se encarrega de eliminar os processos zumbis, liberando seus recursos.

4. Escalonamento de Processos no XV6

O escalonador abordado no XV6 é do tipo Round-robin convencional, isso significa que a tarefa do escalonador é ficar percorrendo a tabela de processos, de forma circular, a procura de processos prontos para execução (i.e. o estado é RUNNABLE) e caso encontre, vai selecionando-os para execução com mesma fatia de tempo para cada processo. Como o XV6 tem múltiplas CPU's e cada CPU tem seu próprio escalonador, quando um escalonador está trabalhando na tabela de processos, a tabela é bloqueada para que não ocorra situações de impasse.

O escalonador do XV6 possui o seguinte algoritmo:

```
void scheduler(void) {
        struct proc *p;
        for(;;){
                sti();
                acquire(&ptable.lock);
                for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
                        if(p->state != RUNNABLE)
                                continue;
                        proc = p;
                        switchuvm(p);
                        p->state = RUNNING;
                        swtch(&cpu->scheduler, proc->context);
                        switchkvm();
                        proc = 0;
                }
                release(&ptable.lock);
        }
}
```

5. Stride scheduling

As características do escalonador de processos Stride Scheduling (escalonamento em passos largos) são semelhantes ao escalonador Lottery Scheduling (escalonador por loteria) em que cada processo recebe um número de bilhetes (tickets), no entanto, ao invés do processo ser sorteado para ser executado, como acontece no escalonador por loteria, no Stride Scheduling os processos recebem um "passo" que é definido com a divisão de um número constante pela quantidade de bilhetes de cada processo, o que faz com que cada processo ganhe a CPU com uma frequência proporcional ao número de tickets.

Inicialmente quando um processo for iniciado, sua passada seré igual a 0 e cada vez que ele ganha a CPU sua passada é incrementada com o valor do seu passo. O escalonador seleciona um processo para ocupar a CPU através de sua passada escolhendo aquele processo que possui o menor valor. Como pode haver situações em que mais de um processo possui a mesma passada, como por exemplo na primeira execução, em que todos os processos tem a passada igual a 0, ele pode escolher um processo qualquer de acordo com a implementação, um exemplo é de acordo com o ID do processo.

Por exemplo, dados os processos A, B e C, cada um com 50, 100 e 200 tickets respectivamente e o valor constante igual a 10000, realizando a operação de divisão obtém-se 200, 100 e 50 que são os valores dos passos dos processos A, B e C respectivamente. Utilizando a ordem lexicográfica como critério de desempate, A executará por primeiro, tendo sua passada atualizada para 200, depois B é executado e sua passada é incrementada para 100, por fim, executa-se C, que terá sua passada atualizada para 50, de acordo com a política, o processo a ser executado a seguir é o processo C, pois possui a passada com menor valor, dessa maneira, a seleção a feita enquanto tiver processos na lista de pronto. Na implementação que abordamos, em situações de empate, permanece selecionado para execução o primeiro processo de menor stride encontrado.

6. Modificações no XV6 para o Projeto

Para a implementação do escalonador *Stride Scheduling*, no XV6 foi necessário estar efetuando algumas modificações nos arquivos fonte do sistema operacional. Os arquivos modificados, bem como sua modifição, serão descritos a seguir.

6.1. Makefile e Teste

Para verificar o correto funcionamento das mudanças no XV6, foi criado um arquivo de teste que cria 3 processos. Tais processos receberão um número diferente de tickets, passados através da função fork, como mostrado a seguir.

```
#include "types.h"      //para usar alguns tipos de variaveis
#include "user.h"       //para usar as funcoes printf e syscalls
#include "param.h"
#include "stat.h"

#define MAX_TESTE 5000
#define MAX_PROCESS 3

void teste(){
  int i, j;
    for(i=0; i<MAX_TESTE; i++)
      for(j=0; j<MAX_TESTE; j++);
}

int main() {
  int i, vet_pid[MAX_PROCESS];

  for(i=1; i<=MAX_PROCESS; i++) {

    if(i==1){
      vet_pid[i]=fork(200);
    }else if(i==2){
      vet_pid[i]=fork(25);
    }else{
      vet_pid[i]=fork(50);
    }

```

```
    if (vet_pid[i] == 0){
      teste();
      exit();
    }

  }

  for(i=0; i<MAX_PROCESS; i++)
    wait();
  exit();

  return 0;
}
```

Para executar o teste dentro do XV6, foram adicionados os comandos de execução do programa no Makefile.

```
    _sh\
    _stressfs\
    _usertests\
    _test\          //esta linha foi adicionada
    _wc\
    _zombie\
    ...

 CPUS := 1          //para fins de teste, alterado de 2 para 1
 endif
 QEMUOPTS = -hdb fs.img xv6.img -smp $(CPUS) -m 512 $(QEMUEXTRA)
@@ -240,7 +241,7 @@ qemu-nox-gdb: fs.img xv6.img .gdbinit
 EXTRA=\            //adicionado test.c
 mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
 ln.c ls.c mkdir.c rm.c stressfs.c usertests.c test.c wc.c
 ...
```

6.2. Proc.h

Neste arquivo foi modificada a estrutura dos processos, sendo adicionadas duas variaveis que serão o passo e a passada.

```
...
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int stride;                  // Valor da passada do processo
  int step;                    // Valor do passo do processo
```

6.3. Proc.c

Iniciamos a implementação do escalonador projetando o seu comportamento, o modo como a ”passada” e o ”passo” são definidos e como ele percorre a tabela de processos procurando o que tem o menor passo.

```
...
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->stride=0;                // Seta uma passada inicial
  if(tickets<=0){             // Seta um numero default de tickets
    tickets=DEF_TICKETS;      // caso nao haja tickets
  }
  p->step=CONSTANT/tickets; // Seta o passo do processo
  release(&ptable.lock);

  ...

void scheduler(void) {
  int stride;
  struct proc *p, *m;      //Adiciona um ponteiro auxiliar para
                           //percorrer a tabela de processos

  while(1) {
    sti();

    acquire(&ptable.lock);
    stride = MAX_STRIDE;
    p = 0;
    //acha o processo com o menor stride
    //percorre a tabela de processos e para cada processo
    for(m = ptable.proc; m < &ptable.proc[NPROC]; m++) {
    //verifica se o processo estÃ¡ em estado runnable e se tem a
        menor passada
      if((m->state == RUNNABLE) && (m->stride < stride)) {
        //se tiver o menor passo, stride recebe sua passada
        stride = m->stride;
        //ponteiro p recebe o processo
        p = m;
      }
    }

    //verifica se existe algum processo na tabela de processos
    if(p){
      //cprintf(" %s,     %d,     %d,     %d", p->name, p->pid,
         p->step, p->stride);
      //se exixtir soma-se a passo do processo a sua passada
         atual
      p->stride += p->step;
      //cprintf("       %d\n", p->stride);
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
```

```
47        //cprintf(" %s,     %d,     %d,     %d, %d\n", p->name, p
              ->pid, p->step, p->stride);
48
49        // Process is done running for now.
50        // It should have changed its p->state before coming back.
51        proc = 0;
52      }
53
54      release(&ptable.lock);
55    }
56  }
```

6.4. Param.h

Neste arquivo foram definidos os valores de uma constante para fazer a divisão e obter o passo de cada processo, um constante auxiliar para selecionar o processo com o menor passo e um número default de tickets que um processo recebe caso nao seja informado no momento de sua criação.

```
1  ...
2  #define FSSIZE       1000  // size of file system in blocks
3  #define CONSTANT         10000
4  #define MAX_STRIDE       99999999
5  #define DEF_TICKETS      1000
```

6.5. Função fork()

Nos arquivos: *defs.h, proc.c e user.h*, a inicialização da função *int fork(void)* foi alterada para receber como parâmetro, um inteiro correspondente ao número de bilhetes que o processo recebe:

```
1  int fork(int);
```

Assim consequentemente precisamos alterar os arquivos onde a função *fork()* é chamada, passando para a função o número de bilhetes que o processo receberá. Segue os arquivos e as alterações:

forktest.c, init.c, sh.c, teste.c, usertests.c

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

stressfs.c

```
1  if(fork(0) > 0)
2  break;
```

sysproc.c

Esta função retorna para quem a chamou, o retorno da função fork().

```
1  int sys_fork(void) {
2    int argtickets;
3    argint(0,&argtickets);
4    return fork(argtickets);
5  }
```

zombie.c

```
int main(void) {
  if(fork(0) > 0)
    sleep(5);
  exit();
}
```

7. Teste, Resultados e Conclusão

Como dito anteriormente, para testar se o comportamento do escalonador era o esperado, utilizamos o arquivo test.c. São criados 3 processos, que são identificados como 4, 5 e 6. O processo 4 recebe 200 tickets e terá passada de 50, o processo 5 recebe 25 tickets e terá passada 400 e o último processo recebe 50 tickets e terá passada 200. O processo 3 que é o pai desses processos recebe o número default de tickets (1000) e terá uma passada igual a 10. Apartir disso podemos observar a distribuição de CPU que cada um deles recebe ao longo do tempo. *Na Figura 1* é possivel observar o padrão desta distribuição.

8. Anexos

Figura 1: Teste e Resultados

proc	Pid	PassadaA	Passo	PassadaP
sh,	3,	10,	70	80
test,	3,	10,	80	90
test,	4,	50,	0	50
test,	4,	50,	50	100
test,	3,	10,	90	100
test,	3,	10,	100	110
test,	5,	400,	0	400
test,	4,	50,	100	150
test,	3,	10,	110	120
test,	6,	200,	0	200
test,	4,	50,	150	200
test,	4,	50,	200	250
test,	6,	200,	200	400
test,	4,	50,	250	300
test,	4,	50,	300	350
test,	4,	50,	350	400
test,	4,	50,	400	450
test,	5,	400,	400	800
test,	6,	200,	400	600
test,	4,	50,	450	500
test,	4,	50,	500	550
test,	4,	50,	550	600
test,	4,	50,	600	650
test,	6,	200,	600	800
test,	4,	50,	650	700
test,	4,	50,	700	750
test,	4,	50,	750	800
test,	4,	50,	800	850
test,	5,	400,	800	1200
test,	6,	200,	800	1000
test,	4,	50,	850	900
test,	3,	10,	120	130
test,	3,	10,	130	140
test,	6,	200,	1000	1200
test,	5,	400,	1200	1600
test,	6,	200,	1200	1400
test,	6,	200,	1400	1600
test,	5,	400,	1600	2000
test,	6,	200,	1600	1800
test,	6,	200,	1800	2000

tickets	1000	200	25	50
passo	10	50	400	200
processo	3	4	5	6
inicio	80(fork do sh)	X	X	X
	90	0	X	X
	90	50	X	X
	90	100	X	X
	100	100	X	X
	110	100	0	X
	110	100	400	X
	120	100	400	0
	120	100	400	200
	120	150	400	200
	120	200	400	200
	120	250	400	200
	120	250	400	400
	120	300	400	400
	120	350	400	400
	120	400	400	400
	120	450	400	400
	120	450	800	400
	120	450	800	600
	120	500	800	600
	120	550	800	600
	120	600	800	600
	120	650	800	600
	120	650	800	800
	120	700	800	800
	120	750	800	800
	120	800	800	800
	120	850	800	800
	120	850	1200	800
	120	850	1200	1000
	120	900	1200	1000
	130	MORTO	1200	1000
	140	MORTO	1200	1000

Execução do teste, mostrando a distribuição de CPU para cada processo. PassadaA é a passada antes de ser somado o passo. PassadaP é a passada posterior a soma do passo.

9. Referências

Para o melhor compreendimento do funcionamento do XV6 tanto quanto seu código fonte foi necessária a utilização de dois livros [Arpaci-Dusseau and Arpaci-Dusseau 2015] e [Cox et al. 2012].

References

Arpaci-Dusseau, R. H. and Arpaci-Dusseau, A. C. (2015). *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.90 edition.

Cox, R., Kaashoek, F., and Morris, R. (2012). ource code: Xv6 a simple, unix-like teaching operating system. MIT, 7 edition.