Implementação e Análise da Política de Escalonamento Stride Scheduling no XV6

Eliton Traverssini[1], **Igor Beilner**[1]

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

{eliton.traverssini, igor.beilner}@gmail.com

Abstract. *The XV6 is a simple Unix-like developed by academics from MIT for teaching purposes. This implementation has a conventional scheduler Round-robin. The objective of this study is to analyze and describe how the management processes in XV6 (i.e. creation, allocation, scheduling and finishing processes) and then implement the Stride Scheduling policy. We will present a set of metrics that will make a basis for the analysis of the functioning of this policy.*

Resumo. *O XV6 é um simples Unix-like desenvolvido por acadêmicos do MIT para fins didáticos. Esta implementação possui um escalonador Round-robin convencional. O objetivo deste trabalho é analisar e descrever como funciona o gerenciamento de processos no XV6 (i.e. criação, alocação, escalonamento e finalização de processos) e então, implementar a política de escalonamento Stride Scheduling. Apresentaremos um conjunto de métricas que servirão de base para a análise do funcionamento dessa política.*

1. O XV6

O XV6 é um sistema operacional para fins didáticos, cujo sistema é uma reimplementação do Unix Version 6 (v6) de Dennis Ritchie e Ken Thompson (1975). O estruturamento deste sistema segue o estilo do v6, porém é voltado para um sistema mais moderno, baseado na plataforma Intel x86, com múltiplos processadores [Cox et al. 2012b]. O XV6 foi desenvolvido no MIT por acadêmicos de Engenharia de Sistemas Operacionais em 2006.

2. Gerenciamento de Processos

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 o seu próprio, mantendo segurança entre os processos. O kernel associa um identificador *PID* de cada processo, de modo que o sistema possa identificar cada processo individualmente, já que o *PID* é único para cada processo. O kernel, também, mantém atualizado o estado de execução que o processo encontra-se (SLEEPING, RUNNING, RUNNABLE, UNUSED, ZOMBIE ou EMBRYO). Quando um processo solicita algum recurso privado do kernel, via chamada de sistema (*system call*), o contexto de execução do processo é desviado ao kernel, que executa o pedido e, após, retorna para o segmento do processo. Um processo em estado de *RUNNABLE* é selecionado pelo escalonador para entrar em execução tão logo quando a CPU estiver disponível.

O XV6 utiliza tabelas de páginas para dar a cada processo seu próprio espaço de endereçamento. As tabelas de páginas mapeiam um endereço virtual para um endereço físico que será enviado pelo processador para a memória principal. O espaço de endereçamento de um processo mapeia instruções e dados do kernel, e também a memória do programa do usuário. 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 de I/O, mas o recurso não está disponível, o processo perde a CPU e tem seu estado alterado para SLEEPING.

Os processos do XV6 possuem em sua estrutura um vetor *inode*, que indica os arquivos abertos por este processo. Cada *inode* tem um número que indica sua posição no disco. O kernel mantém na memória os *inodes* que estão em uso pelos múltiplos processos.

2.1. Criação de Processos

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 *int fork()*, o processo criado é chamado processo filho, pois é uma cópia do processo que invocou a chamada *int fork()*. O processo filho, também, herda do processo pai o mesmo conteúdo de memória, contudo que cada processo tem seu espaço de endereçamento distinto. O XV6 aloca memória do espaço do usuário implicitamente. A chamada *int fork()* aloca memória sufuciente 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 intruções 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.

2.2. Escalonamento de Processos

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. Na próxima seção, apresentaremos a proposta de escalonamento que foi implementada e avaliada no XV6. Essa proposta traz uma abordagem diferente, pois agrega prioridade entre os processos, dando mais fatias de tempo de CPU para os processos de alta prioridade e, consequentemente, menos fatias de tempo de CPU para os processos de menor prioridade.

3. Proposta de Escalonador

A política de escalonamento de processos implementada é a *Stride Scheduling* (escalonamento em passos largos) [Arpaci-Dusseau and Arpaci-Dusseau 2015], esta política é bastante semelhante à de loteria, em que é dado um certo número de bilhetes (tickets) a cada processo, no entanto, ao invés do processo ser sorteado para ser executado, como na política de loteria, no escalonamento em passos largos, é definido o passo de cada processo, que é encontrado com 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 tikets muito rapidamente.

Quando um processo é iniciado, sua passada é igual a 0 e cada vez que ele ganha a CPU sua passada é incrementada com o valor do seu passo, sempre que o escalonador escolhe um processo para ocupar a CPU, ele seleciona o processo que tem a menor passada. Como podem 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 tickes 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.

3.1. Estruturas de Dados

A seguir são apresentadas as estruturas de dados [Cox et al. 2012a] e funções responsáveis por representar, criar e escalonar processos a serem executados na CPU.

```
struct proc {
    uint sz;                     //tamanho do processo na memoria
    pde_t* pgdir;                //tabela de paginas
    char *kstack;                //pilha de kernel do processo
    enum procstate state;        //estado do processo
    volatile int pid;            //ID do processo
```

```
    struct proc *parent;         //ponteiro para o pai do processo
    struct trapframe *tf;        //frame de interrupção
    struct context *context;     //troca de contexto para
    void *chan;                  //se não e zero esta dormindo
    int killed;                  //se não e zero foi morto
    struct file *ofile[NOFILE]; //abertura de arquivos
    struct inode *cwd;           //diretório corrente
    char name[16];               //nome do processo
    int stride;                  //valor da passada do processo
    int step;                    //valor do passo do processo
};
```

A estrutura *struct proc* é responsável por representar um processo, além de seus atributos originais, foi necessário a inclusão de outros dois: *int stride* e *int step*. O atributo *step* é responsável por representar o passo do processo, já *stride* serve para controlar a passada de cada processo, sendo esta incrementada com o valor de *step* a cada vez que o processo ganha a CPU.

```
int fork(int tickets) {
    int i, pid;
    struct proc *np;

    // Allocate process.
    if((np = allocproc(tickets)) == 0)
        return -1;

    // Copy process state from p.
    if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    np->sz = proc->sz;
    np->parent = proc;
    *np->tf = *proc->tf;

    // Clear %eax so that fork returns 0 in the child.
    np->tf->eax = 0;

    for(i = 0; i < NOFILE; i++)
        if(proc->ofile[i])
            np->ofile[i] = filedup(proc->ofile[i]);
    np->cwd = idup(proc->cwd);

    safestrcpy(np->name, proc->name, sizeof(proc->name));
```

```
    pid = np->pid;

    // lock to force the compiler to emit the np->state write last.
    acquire(&ptable.lock);
    np->state = RUNNABLE;
    release(&ptable.lock);

    return pid;
}
```

A função *int fork()*, que é responsável por criar os processos, sofreu modificação no seu parâmetro, recebendo a quantidade de *tickets* atribuídos a cada processo. A proposta inicial, apresentada no trabalho 1, era inicializar os atributos *step* e *stride* nesta função, porém como o primeiro processo do sistema é criado atrevés da função *userinit* e o restante dos processos são criados pela função *int fork()*, mas ambas funções alocam os processos na função *proc * allocproc()*, a inicialização dos atributos foi realizada nesta função, conforme o fragmento de código a seguir:

```
static struct proc* allocproc(int tickets) {
    struct proc *p;
    char *sp;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == UNUSED)
            goto found;
    release(&ptable.lock);
    return 0;

found:
    p->state = EMBRYO;
    p->stride = 0;                //inicialização da passada
    if(tickets <= 0)
        tickets = DEF_TICKETS;    //inicialização default de tickets
    p->step = CONSTANT/tickets;   //inicialização do passo
    p->pid = nextpid++;
    release(&ptable.lock);

    // Allocate kernel stack.
    if((p->kstack = kalloc()) == 0){
        p->state = UNUSED;
        return 0;
    }
    sp = p->kstack + KSTACKSIZE;

    // Leave room for trap frame.
    sp -= sizeof *p->tf;
    p->tf = (struct trapframe*)sp;
```

```
    // Set up new context to start executing at forkret,
    // which returns to trapret.
    sp -= 4;
    *(uint*)sp = (uint)trapret;

    sp -= sizeof *p->context;
    p->context = (struct context*)sp;
    memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;

    return p;
}
```

A função *void scheduler()* é responsável por escalonar processos que concorrem pela CPU. O fragmento de código a seguir apresenta a implementação da política *Stride Scheduling*, cuja execução consiste em percorrer a tabela de processos em busca de um processo com estado *RUNNABLE* de menor *stride*, esta escolha ocorre pela comparação do *stride* do processo com o anterior, verificando se este é menor (i.e em caso de empate o anterior permanece). Além disso, é necessário verificar se um processo realmente foi selecionado, pois pode ocorrer de nenhum processo estar *RUNNABLE*.

```
void scheduler(void) {
    int stride;
    struct proc *p, *m;

    while(1) {
        // Enable interrupts on this processor.
        sti();
        acquire(&ptable.lock);
        stride = MAX_STRIDE;    // parâmetro inicial de passada
        p = 0;
        for(m = ptable.proc; m < &ptable.proc[NPROC]; m++) {
            if((m->state == RUNNABLE) && (m->stride < stride)) {
                stride = m->stride;
                p = m;
            }
        }

        if(p){  //verifica se algum processo foi selecionado
            p->stride += p->step;
            proc = p;
            switchuvm(p);
            p->state = RUNNING;
            swtch(&cpu->scheduler, proc->context);
            switchkvm();
            proc = 0;
        }
```

```
            release(&ptable.lock);
        }
    }
```

4. Métricas de Avaliação

Para avaliar se o escalonador foi implementado de maneira correta foram criados vários processos iguais com uma conhecida distribuição de *tickets* entre eles, esses processos concorrem pela CPU para executarem suas tarefas. A implementação é validada observando a sequência de término dos processos.

5. Resultados Obtidos

A realização dos testes para avaliar a corretude da implementação se deu através da criação de dez processos com as respectivas quantidades de *tickets* 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 e foi utilizado o valor constante 10000 para o cálculo do atributo *step*.

Cada processo consiste na execução de dois laços aninhados, cada um iterando de 0 a 9000. O mesmo teste foi repetido vinte vezes a fim de dar credibilidade nos resultados. Os resultados obtidos são apresentados na Tabela 1.

Ordem	Name	PID	Step	Stride	CPU
1	schedulertests	13	1111	136653	123
2	schedulertests	14	1000	141000	141
3	schedulertests	12	1250	163750	131
4	schedulertests	11	1428	202776	142
5	schedulertests	10	1666	261562	157
6	schedulertests	9	2000	278000	139
7	schedulertests	8	2500	315000	126
8	schedulertests	7	3333	486618	146
9	schedulertests	6	5000	615000	123
10	schedulertests	5	10000	1430000	143

Tabela 1. Ordem de finalização dos processos

6. Conclusão

Observando os resultados obtidos, conclui-se que a implementação se deu de maneira satisfatória, condizendo com os resultados esperados, pois de acordo com a Tabela 1, nota-se que os processos encerraram sua execução em ordem proporcional ao atributo *step*, que implica na quantidade de *tickets*.

A execução da política do *Stride Scheduling*, em comparação com o *Round-robin*, da forma como implementamos, se tornou mais complexa. Isso quer dizer que há maior processamento para o escalonamento de processos neste novo modelo, pois toda vez o escalonador percorre a lista de processos até o final para escolher um processo *RUNNABLE* e com menor *stride*, ou seja, a complexidade do *Stride Scheduling* será sempre linear no número de processos. No caso do *Round-robin* esse processamento é dispensado, pois o escalonador pegará o primeiro processo *RUNNABLE*, que no melhor caso será *O(1)* e no pior, *O(n^o processos)*. A questão abordada pelo *Stride Scheduling* é saber qual processo

deve-se executar primeiro, baseando-se nas prioridades de cada processo, diferentemente do *Round-robin*, cujo escalonamento é feito sob a ordem de chegada dos processos, selecionando o primeiro processo *RUNNABLE*, apenas.

Uma sugestão de trabalho futuro seria a implementação desta política de escalonamento utilizando *Heaps Binárias*, o que faria com que a execução do escalonamento fosse otimizada, deixando-a com complexidade $O(log(n))$, sendo n o número de processos.

Referências

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. (2012a). *Source Code: XV6 a simple, Unix-like teaching operating system*. MIT, 7 edition.

Cox, R., Kaashoek, F., and Morris, R. (2012b). *XV6 a simple, Unix-like teaching operating system*. MIT, 7 edition.