Implementação do Stride Scheduler no sistema operacional Xv6

Nicholas Sangoi Brutti[1], **Ricardo Augusto Müller**[1]

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

{nicholassbrutti,ricardoam0908}@gmail.com

Abstract.

Resumo. *Este artigo descreve uma análise do processo de escalonamento padrão utilizado no sistema operacional Unix-like Xv6. A seguir uma outra abordagem de escalonamento é proposta, trata-se do Stride Scheduler, um escalonador proporcional determinístico. O trabalho também contém as etapas necessárias para a implementação, os testes efetuados e resultados obtidos.*

1. Introdução

O sistema operacional é um software responsável por gerenciar diversos recursos (i.e. processador, discos, memória principal, dispositivos de E/S) com a finalidade de apresentar um modelo computacional simples e que facilite ao programador e usuário final em suas atividades. Como o sistema operacional é a peça mais básica de software ele opera em modo núcleo, ou seja, possui acesso completo a todo o hardware. O restante opera em modo usuário, na qual o acesso é restrito a um subconjunto de instruções [Tanenbaum 2010]. As requisições de acesso a recursos privilegiados é efetuada através de chamadas de sistema (***system call***).

Como em computadores multiprogramados existem diversos processos e *threads* disputando acesso à CPU, é preciso que exista um critério de escolha para determinar qual executar. Nesse contexto, existe o escalonador que define qual tarefa será atendida primeiro. Atualmente, existem vários algoritmos de escalonamento, porém, esse estudo está voltado somente para o Stride Scheduler (escalonador por passos largos).

2. O Xv6

Trata-se de um sistema operacional de código aberto escrito na linguagem C, desenvolvido por acadêmicos do MIT *(Massachusetts Institute of Technology)*. O Xv6 é uma implementação do antigo Unix versão 6 na arquitetura Intel x86 [Cox et al. 2014]. Devido suas limitações ele é utilizado geralmente como uma ferramenta de apoio no aprendizado de sistemas operacionais. Porém, como a base é Unix alguns conceitos presentes em sistemas operacionais mais robustos são herdados.

2.1. Processos e memória

Como o Xv6 utiliza a arquitetura de kernel monolítico (assim como a maioria dos sistemas operacionais conhecidos), existe uma definição de privilégios para execução de processos. Quando um programa de usuário (com nível de privilégio padrão) passa possuir chamadas

de sistema (*system call*) o mesmo precisa ser executado em modo kernel para assim garantir a proteção e consistência dos dados, sendo assim, a CPU é responsável por realizar a troca de contexto entre o modo kernel e usuário.

Como exemplo mais evidente nas alterações que seguirão nesse projeto, a função *fork()* do arquivo *proc.c* trata-se de um desses casos descritos acima, pois o mesmo é chamada de sistema e é executada dentro do kernel.

2.1.1. Função fork()

É a função responsável por instanciar *(fork)* um novo processo, chamando a função *allocproc()*, que aloca a memória do processo, e deixa o processo em estado RUNNABLE, ou seja, pronto para ser selecionado pelo escalonador. A função fork possui como retorno o PID (identificador único do processo), que é passado para o processo que fez a chamada da função, também conhecido como o processo pai.

2.1.2. Função allocproc()

Sua função principal é a alocação de um novo processo na memória, para tal, o allocproc varre a lista de processos *(ptable.proc)* e busca por uma posição que possua um estado de inutilização, quando a encontra, as informações do processo são atribuídas à estrutura e o estado do processo é setado para EMBRYO, que trata-se do estado recém alocado e ainda não pronto para ser selecionado pelo escalonador. O retorno da função allocproc é um ponteiro para o então novo processo alocado.

2.1.3. Funções complementares dos processos

Faz-se necessário a citação de algumas funções importantes para a manipulação dos processos, sendo elas, a função *exit()*, *kill()*, *wait()* e *sleep()*, sendo as duas primeiras, funções que tratam do término de um processo. Quando um processo ocorre como o esperado e é encerrado, a função *exit()* é executada e o estado do processo é setado para ZOMBIE, em contraponto, quando ocorre uma saída abrupta (evento não esperado), executa-se a função *kill()*.

A função *wait()* tem como objetivo forçar o processo pai a aguardar o término de todos os seus processos descendentes, enquanto a função *sleep()* faz com que qualquer processo (seja este pai ou filho) pare sua execução e aguarde por um determinado período de ciclos de clock.

2.2. O Escalonador do Xv6

O escalonamento no Xv6 por padrão utiliza o algoritmo *round-robin*, que basicamente, percorre um vetor de processos (*ptable.proc*) até encontrar um processo que esteja no estado *RUNNABLE* (pronto), e então esse processo é alterado para o estado *RUNNING* (executando) e ele parte para execução. A cada processo é atribuído um intervalo de tempo (*quantum*), no qual ele é permitido executar. Caso o processo exceda seu tempo ele é bloqueado e a CPU sofrerá preempção para selecionar outro processo. É possível

notar que nesse tipo de escalonamento não há como estabelecer prioridade de execução, o escalonador seleciona o primeiro que ele encontrar em estado RUNNABLE. A próxima seção apresenta o escalonador por passos largos (*Stride Scheduling*) que tem como diferencial a definição da prioridade do processo.

3. Escalonamento em passos largos (*Stride Scheduling*)

O escalonamento em passos largos consiste em atribuir a cada processo uma quantidade de bilhetes(tickets), que em outras palavras, representa o grau de prioridade. Ao contrário do escalonamento por loteria caracterizado pela abordagem probabilística, onde a disputa acontecia através de um sorteio, nesse caso não há aleatoriedade, o processo selecionado é o que possui o valor mínimo da passada, isso caracteriza uma abordagem determinística [Waldspurger and Weihl 1995]. Primeiramente, calcula-se o ”passo”do processo como sendo o resultado da divisão de constante (ex. 10000) pelo número de tickets do processo. O escalonador então efetua a busca do processo que contenha a menor ”passada”, inicialmente todos possuem 0, e esteja disponível para executar (estado = RUNNABLE), e então o acesso à CPU é concedido. Logo após ser selecionado, a ”passada”do processo é incrementada com o valor do passo, e o escalonador prossegue sua execução. Todos esses dados foram adicionados na estrutura do processo, conforme demonstrado abaixo.

```
// Per-process state
struct proc {
    uint sz;
    pde_t* pgdir;
    char *kstack;
    enum procstate state;
    int pid;
    struct proc *parent;
    struct trapframe *tf;
    struct context *context;
    void *chan;
    int killed;
    struct file *ofile[NOFILE];
    struct inode *cwd;
    char name[16];
    int tickets;
    int position;
    unsigned long int stride;
};
```

Código-fonte 1: Estrutura de um processo. Trecho retirado do arquivo proc.h.

3.1. Implementação

Conforme mencionado, ao criar um processo é preciso que seja informado a quantidade de bilhetes que o mesmo deve possuir. Dessa forma, uma alteração foi realizada na estrutura do processo (*proc.h*), onde foi adicionado uma variável responsável por salvar essa informação e permitir a consulta. A quantidade de bilhetes é repassada como parâmetro para a função *allocproc(int tickets)*, que por sua vez, é chamada através da função *fork(int tickets)*. Antes da inserção na estrutura é verificado se a quantidade solicitada está entre o intervalo permitido, caso a quantidade q seja menor que 0 (*MINTICKET*) ele recebe

baixa prioridade (*LOWPRIOR*), e caso seja maior que 4096 (*MAXTICKET*) ele recebe prioridade alta (*HIGHPRIOR*). As constantes estão definidas no arquivo *param.h*.

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

Código-fonte 2: Constantes adicionadas *sysproc.c*.

Consequentemente isso resultou na alteração de todas as chamadas da função *fork()*, pois antes não havia a necessidade de informar a quantidade de bilhetes. Os arquivos modificados foram o *sysproc.c* na função *sys_fork()*.

O Xv6 possui uma função *scheduler()* no arquivo *proc.c* que implementa o escalonador. Foi preciso intervir na métrica utilizada para escolha do processo, conforme descrito na seção 3.

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

    //aux to decide to which process the ticket belong
    unsigned long int menor = 1234567891;

    acquire(&ptable.lock);
    select = ptable.proc;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state == RUNNABLE && p->stride < menor){
            menor = p->stride;
            select = p;
        }
    }

    checkStride(ptable.proc, select);
    select->stride += (STRIDE_CONSTANT/select->tickets);
    c->proc = select;
    switchuvm(select);
    select->state = RUNNING;
    swtch(&(c->scheduler), select->context);
    switchkvm();

    c->proc = 0;
    release(&ptable.lock);

}
```

Código-fonte 3: Implementação do escalonador *Stride Scheduling*

A função *checkStride(struct proc *, struct proc *)* tem como finalidade evitar que ocorra overflow na variável stride, pois embora seja uma variável do tipo unsigned long

int pode haver situações onde o stride ultrapasse o valor permitido, o que acarretaria inconsistências no escalonamento.

4. Análise de desempenho

Essa solução resolve o problema, porém não é a mais eficiente. Para encontrar o processo com a menor passada no pior caso uma varredura completa deverá ser realizada no vetor de processos, complexidade $O(n)$. Uma outra solução seria adotar uma estrutura de dados auxiliar que contenha todos os processos que estão em estado pronto. Essa estrutura seria uma *min-heap*, uma árvore implementada sob um vetor, cuja sua principal característica é a consulta constante $O(1)$, pois se respeitada as restrições o menor valor encontra-se na raiz da árvore. Como as operações de inserção e remoção podem violar as condições da estrutura, existe um procedimento que busca realocar as posições de tal forma que a estrutura permaneça confiável, conhecido como *heapfy*, com um custo $O(log\ n)$.

Uma outra alternativa, é efetuar a ordenação do vetor de processos de forma crescente no valor da passada. Porém torna-se impraticável porque dentre os vários algoritmos existentes, o mais rápido possui complexidade $O(n\ log\ n)$ no pior caso, onde os vetores já encontram-se ordenados. Este caso pode ser recorrente em um escalonador do tipo Stride Scheduler, sendo assim, é mais vantagem percorrer o vetor de processos de forma linear.

5. Testes e resultados

Para testar o funcionamento da solução foram desenvolvidos quatro programas (*teste1*, *teste2*, *teste3* e *teste4*). Para que possa ser medida a funcionalidade do sistema de escalonamento, processos que apenas consomem CPU foram criados, todos eles com complexidade n^3, com n variando em cada teste e todos estes foram adicionados ao Makefile do Xv6.

O *teste1* realiza a criação de 5 processos com as respectivas quantidades de bilhetes: 1024, 128, 512, 64, 2048 com n igual a 600. As quantidades de bilhetes foram definidas de forma ”aleatória”, mas mantendo uma proporção, para testar o funcionamento do sistema. Os processos encerram-se como esperado, na ordem: 2048, 1024, 512, 128, 64.

O *teste2* realiza a criação de 10 processos com as respectivas quantidades de bilhetes: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 com n igual a 600. As quantidades de bilhetes foram definidas de forma crescente. Os processos encerram-se como esperado, na ordem: 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2.

O *teste4* realiza a criação de 4 processos com as respectivas quantidades de bilhetes: 3, 4, 5, 6 com n igual a 900. As quantidades de bilhetes foram definidas de forma crescente, mas com valores de tickets bem próximos. Os processos encerram-se como esperado, na ordem: 6, 5, 4, 3.

6. Conclusão

Pode-se concluir através dos testes efetuados que a implementação do *Stride Scheduler* obteve um resultado satisfatório, o número de tickets influenciou de forma proporcional ao tempo adquirido de CPU. Por isso torna-se muito útil para situações de compartilhamento de recursos onde é preciso dedicar uma porcentagem da CPU para um processo específico.

Referências

Cox, R., Kaashoek, F., and Morris, R. (2014). xv6 a simple, unix-like teaching operating system.

Tanenbaum, A. S. (2010). *Modern operating systems*. Pearson Prentice Hall, 3th edition.

Waldspurger, C. A. and Weihl, W. E. (1995). Stride scheduling: Deterministic proportional-share resource management.