pseudo-SO: Um estudo da gerência de processos, memória e entrada/saída

João Pedro Assis dos Santos
17/0146367
Universidade de Brasília
joaopedroassisdossantos@gmail.com

Pedro Augusto Ramalho Duarte
17/0163717
Universidade de Brasília
pedro_a2312@hotmail.com

Waliff Cordeiro Bandeira
17/0115810
Universidade de Brasília
waliffcordeiro@gmail.com

***Abstract*—O presente trabalho utiliza a ideia de um SO simplificado multiprogramado, composto por um Gerenciador de Processos, Gerenciador de Memória e Gerenciador de Entrada/Saída. Para cada módulo foram implementados, através da linguagem GoLang, três algoritmos, a fim didático e de estudo, que serão descritos e explicados ao longo do relatório do projeto realizado.**

Palavras-chave: ***Sistemas Operacionais, SO, Gerência de Memória, Gerência de Processos, Gerência de IO.***

I. Introdução

Os sistemas operacionais possuem como principais objetivos a gerência de recursos do sistema e proporcionar uma interface/transparência entre o sistema e o usuário. Tendo como base os conceitos estudados na disciplina Sistemas Operacionas da Universidade de Brasília, ministrada pela professora Aletéia, implementamos os módulos de gerência de processos, memória e entrada e saída. Será melhor abordado os três principais algoritmos de cada módulo, no qual faremos a explanação dos conceitos e implementação para um estudo e melhor entendimento das abordagens.

II. Arquitetura

De forma prática, temos a main, que apenas chama o kernel.Exec, passando como argumento qual é o módulo de execução e o arquivo de entrada. O kernel, por sua vez, é responsável por identificar qual módulo precisa ser chamado e executá-lo, passando os argumentos necessários e imprimindo os retornos esperados. No módulo de processos foi implementada uma estrutura de processo que contém o id, tempo de chegada e duração de cada processo e uma estrutura que contém o pID, tempo de início e tempo de fim da execução de cada processo. Essa segunda estrutura foi utilizada em um formato de slice para ser a lista de execução dos processos. No módulo de memória temos a estrutura Memory que contém o tamanho da memória e a sequência de endereços, temos a lista de frames que contém a página e o tempo de chegada e uma terceira estrutura muito semelhante à lista de frames mas adaptada para o caso da segunda chamada, contendo também a referência para determinarmos a segunda chance da página. Por fim, no módulo de IO temos uma estrutura Disk que contém o tamanho do disco, o ponto de partida e a sequência de endereços que ele irá percorrer. Foi uma estrutura simples mas suficiente para implementar todos os algoritmos requisitados.

III. Ferramentas e Linguagens utilizadas

No trabalho utilizamos a linguagem GoLang, na sua versão 1.16.3, com suas bibliotecas padrões. Do ponto de vista de rotina de código utilizamos da prática do Test Driven Development (TDD), ou seja, antes de começar a implementar o código, fizemos testes unitários para os principais algoritmos do nosso trabalho, primeiro tendo o exemplo da especificação como base e posteriormente criando alguns outros casos que possivelmente não eram cobertos por nosso código. Para versionar e organizar as tarefas utilizamos o github, o repositório pode ser encotrado por meio desse link: https://github.com/SwitchDreams/switchOS. Em termos de ambiente de desenvolvimento, dois integrantes utilizarem o VSCODE e um integrante utilizou do GOland.

IV. Módulos

A. Módulo de Gerência de Processos

Neste módulo foram implementados 3 algoritmos de escalonamento de CPU e para cada algoritmo foram calculadas as seguintes métricas: Tempo médio de execução total do processo, Tempo médio de resposta, Tempo médio de espera.

Para executar esses algoritmos, foram implementadas duas estruturas de dados fundamentais, a primeira dela é o *Proccess*, que contém as informações do identificador do processo, o tempo de chegada e a duração. E a segunda é o *ProcessExecution*, que contêm o pid, o tempo de início e termino da execução daquele processo.

```
type Process struct {
        ID          int
        ArrivalTime int
        Duration    int
}

type ProcessExecution struct {
        Pid        int
        StartTime  int
        FinishTime int
}
```

Para explicar melhor o fluxo de código podemos observar melhor o diagrama abaixo:

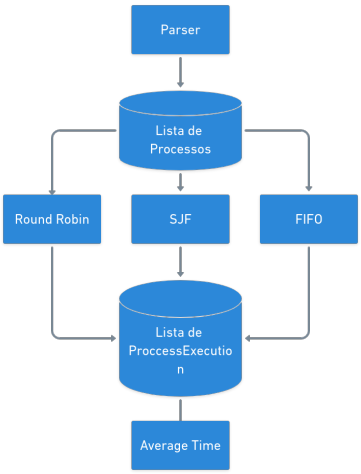

Fig. 1. Digrama do módulo de Processos

1) FIFO: O algoritmo First in First Out mais conhecido como FIFO, simplesmente executa a lista de processos em ordem de chegada e sua implementação é bem simples. No caso da arquitetura, bastou coletar a lista de processos proveniente do *parse*, iterar e criar outra lista de execução de processo em ordem.

2) SJF: O algoritmo Shortest Job First mais conhecido como SJF, se baseia em executar o processo que está mais perto de finalizar dentro os disponíveis naquele instante. Na nossa implementação iteramos a lista de processos em ordem de chegada, coletamos os processos que chegaram na CPU naquele instante e ordenamos em ordem crescente de duração de execução, depois disso pegamos o primeiro termo dessa lista e colocamos na lista de processos executados, por último atualizamos o tempo atual e passamos para próxima iteração até acabar os processos.

3) Round Robin: O algoritmo Round Robin funciona como uma pilha circular, o processador fornece um espaço máximo fixo de tempo para cada processo ser executado que se chama de quantum, se o processo não é finalizado dentro do periodo de um Quantum ele é passado para o final da fila de execução. É o alogirtmo que tem uma implementação mais complexa entre os dois anteriores. Nesse algoritmo no lugar de iterar a lista de processos, faremos um loop com o tempo atual, começamos com o tempo atual de 0 e vamos adicionando a cada ciclo do loop. A primeira parte do loop é coletar os processos que chegaram nesse instante e inserir na lista de processos que devem ser processados pela cpu *registeredProcesses*, também é importante já colocar na lista os processos que vão ser coletados no próximo quantum, para ajustar a ordem da fila quando o processo atual for executado, ou seja, o próximo a entrar na fila do round robin entra antes do que foi executado nesse instante. O próximo passo é adicionar o processo na lista de execução de processos, caso o processo termine antes de encerrar o quantum, na nossa lógica, a cpu já da o espaço para o próximo proceso da lista, e o loop segue até as listas de processos que estão aguardando a execução e a lista de processos que não chegaram fiquem vazias.

4) Average Time: O último algoritmo desse módulo é a função *AverageTime*, que recebe de parametros a lista de processos, a lista de processos executados (output dos algortimos acima) e se o algoritmo é o round robin ou não e ela retorna os 3 indicadores em ordem: Tempo médio de execução total do processo, Tempo médio de resposta e Tempo médio de espera.

Para calcular esses indicadores a função varre a lista de processos executados de trás para frente, verifica se o processo ainda não finalizou e calcula primeiramente o tempo de execução do processo fazendo a subtração do tempo de término do processo e o tempo que ele chegou para CPU, logo depois cálcula o tempo de espera, fazendo a subtração do tempo de execução do processo e a duração do processo, tendo esses valores calculados, cada um é somado em uma variável que armazena o valor total do tempo. Depois que a iteração de processos finaliza, dividimos as somatórias calculadas pelo tamanho da lista de processos. Para calcular o tempo de resposta, caso o algoritmo seja round robin, o tempod e resposta é igualado ao quantum e caso não seja é igualado ao tempo de espera, assim conseguimos obter as 3 métricas, como foi proposto

Vale salientar que em todos os 3 algoritmos lidamos com o fato que não tenha chegado nenhum processo na CPU ainda. E fizemos a exportação da lista de processos para o arquivo como foi proposto no trabalho por meio da função *ExecutionToFile*, que são guardados na pasta *output*

B. Módulo de Gerência de Memória

Neste módudo foram implementados 3 algorítmos de gerência de memória, sendo que cada um retornava o número de faltas presentes durante a execução do algorítmo. Para a correta execução dos métodos, foram implementadas duas estruturas. Primeiramente temos a *Memory*, que armazenava informações sobre o tamanho da mémória total, bem como sua sequência. A outra estrutura se chama *FramesList*, que armazena a página e o momento em que ela chegou.

```
type Memory struct {
        Size     int
        Sequence []int
}

type FramesList struct {
        Page    int
        Arrived int
}
```

1) FIFO: O algorítmo FIFO é o mais simples dos três. Em sua implementação, fazemos um array com o tamanho da memória e inserimos as páginas até que este esteja cheio. A partir daí, a próxima página vai ter que ocupar o espaço de uma que já está alocada. Para isso, tiramos o mod do contador (que conta até o tamanho da sequência de páginas) com o tamanho da memória. Essa operação retorna o índice da localização da

nova página. As faltas são contadas caso a página em questão não esteja na memória e uma substituição seja necessária.

2) LRU: O LRU (menos utilizado recentemente) foi implementado utilizando uma lógica muito simples. Primeiro temos um slice Memory que contém o tamanho e a sequência de memória utilizada e, para auxiliar essa estrutura, temos um slice de frames que contém a página propriamente dita e uma referêcia para sabermos quando ela foi referenciada. Verificamos se o frame está em memória e, se estiver, apenas iremos atualizar a a sua referência, para sabermos se ela foi referenciada a mais ou menos tempo que outro frame. Caso a página não esteja em memória, teremos duas decisões, se houver espaço na memória, apenas iremos inseri-la, se não houver espaço, iremos retirar a página que foi referenciada a mais tempo (menos tempo sem ser referenciada), dando lugar à página que está chegando.

3) Segunda Chance: O algorítmo de segunda chance é o mais complexo dos três. Ele utiliza um bit de referência para cada página armazenada. Caso o espaço para páginas esteja lotado, o algorítmo verifica qual das páginas armazenadas possui o bit de referência igual a zero e, caso ele o encontre, essa será a página removida. Caso o bit seja igual a um, uma contagem é iniciada, onde um contador é incrementado a cada vez que a o bit da página é 1. Ao checar 3 vezes a mesma página o bit é zerado e a próxima vez que esse endereço for referenciado, ele pode ser substituído por outra página

C. Módulo de Gerência de Entrada/Saída

Nesse módulo foram implementados os algorítmos de acesso a disco, sendo que cada um retornava o número de movimentos realizados pela cabeça do disco. Para auxiliar o desenvolvimento, utilizamos a estrutura *Disk*, que armazenava o tamanho total do disco, a posição incial da cabeça e a sequência de posições que deveriam ser acessadas

```
type Disk struct {
        Size     int
        Init     int
        Sequence []int
}
```

1) FCFS: O primerio algorítmo utilizado foi o First Come First Served, que basicamente ia de posição em posição, sem se preocupar com eficiência. Dessa forma, para cada posição da sequência, o algorítmo calculava o módulo da distancia entre a posição atual e a próxima, iterando até **length(Sequence) - 1**, para evitar problemas de indexação com o último valor.

2) SCAN: O algorítmo SCAN utiliza uma abordagem diferente. A partir da posição inicial ele vasculha o disco para todas as posições da esquerda e depois para todas as posições da direita. A cada iteração em cada uma das duas etapas o programa verificava se haviam cilindros com verificação pendendte. Após verificar isso, a posição atual é removida da lista de pendências (que começa com todas as posições a serem verificadas).

3) SSF: O algorítmo Shortest Job Fisrt aplica uma estratégia interessante. Primeiramente, a função inicializa um array auxiliar (que funciona como uma lista de pendencias) para armazenar a sequencia, e inicializa a cabeça com o valor da posição inicial informada. Para cada valor presente na sequencia, a função calcula qual o valor mais próximo dentro da lista de pendências. É calculado módulo da distância da cabeça para o valor mais próximo, e a cabeça agora é o valor atual. O valor atual é removido da lista de pendências para a próxima iteração.

V. Desenvolvimento do trabalho

A arquitetura foi definida em group program através de chamada com screen share. Fizemos o modelo inicial do projeto, implementando o parse e organizando os módulos.

A. Pedro Augusto

Inicialização da arquitetura de cada módulo e o Kernel

1) Gerência de Processos: FIFO, SJF e Função para cálcular médias de tempos
2) Gerência de Memória: FIFO
3) Gerência de Entrada/Saída: SCAN

B. João Pedro

Criação do sistema de erros

1) Gerência de Processos: Round Robin
2) Gerência de Memória: Segunda Chance
3) Gerência de Entrada/Saída: SSF

C. Waliff Cordeiro

Organização do Git

1) Gerência de Processos: SJF e Round Robin
2) Gerência de Memória: LRU e Segunda Chance
3) Gerência de Entrada/Saída: FCFS e SSF

Conclusão

O trabalho realizado na disciplina Sistemas Operacionais foi fundamental para aprofundarmos e entendermos, na prática, os conceitos vistos de forma teórica nas aulas. Aproveitamos para aplicarmos uma abordagem orientada a testes, tentando garantir maior segurança da corretude de todo código. Além da utilização de testes automáticos, escolhemos utilizar a linguagem Golang, uma linguagem pouco conhecida entre os participantes do grupo, a fim de aprender uma nova tecnologia. Julgamos muito importante os conceitos abordados no trabalho, tanto para fins acadêmicos, quanto para fins profissionais em nossa área. Aprender de forma mais prática sobre os conceitos e aplicações da gerência de processos, memória e IO foi muito proveitoso e conseguimos implementar uma boa arquitetura.

References

[1] SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G.;Fundamentos de Sistemas Operacionais, 9ª ed.LTC, 2017.

[2] Google. Golang Documentation. Disponível em: ¡https://golang.org/doc/¿, Acesso em: 11 de maior de 2021.

[3] TANENBAUM, A. S. SistemasOperacionaisModernos, São Paulo: Pearson, 4ª ed., 2015