Relatório - Trabalho Prático de SO

Ana Paula Martins
Departamento de Ciências da Computação
Universidade de Brasília
Brasília, Brasil
17/0056082
aptarchetti@gmail.com

Gabriel Matheus
Departamento de Ciências da Computação
Universidade de Brasília
Brasília, Brasil
17/0103498
gmatheusoliveirasys@gmail.com

Gustavo Carvalho
Departamento de Ciências da Computação
Universidade de Brasília
Brasília, Brasil
17/0058867
gmcmacar@gmail.com

Guilherme Braga
Departamento de Ciências da Computação
Universidade de Brasília
Brasília, Brasil
17/0162290
g_braga_545@protonmail.com

***Resumo*—Este relatório apresenta como objetivo documentar a solução proposta da implementação de um sistema operacional teórico como avaliação da matéria de Sistemas Operacionais na Universidade de Brasília, a fim de testar funcionalidades básicas de um sistema operacional real.**

***Index Terms*—Sistemas Operacionais, Python, *Threads*, Processos, Interrupções, Escalonamento, Entrada e Saída.**

I. Introdução

O objetivo deste trabalho é implementar um pseudo-SO conforme a especificação disponibilizada. O SO deve possuir os módulos de gerência de processos, de gerência de memória e de gerência de entrada e saída. Em primeiro lugar, o módulo de **gerência de processos** deve conseguir simular o gerenciamento de processos na CPU através dos seeguintes algoritmos: FIFO, Shortest Job First e Round Robin. Depois, o módulo de **gerência de memória** deve simular a troca de páginas carregadas em memória através dos algoritmos FIFO, Second Chance e LRU. Por fim, o módulo de **gerência de entrada e saída** deve ser capaz de simular o funcionamento do escalonamento de disco com os algoritmos FCFS, SSF e SCAN.

Na subseção I.A, serão descritas as ferramentas utilizadas na implementação do pseudo-SO, na seção II, será detalhada a implementação de cada um dos gerenciadores em questão, na seção III será discorrido sobre a dificuldades encontradas e as respectivas soluções propostas, na seção IV será relatado o papel de cada aluno na realização do trabalho e a seção V terá as conclusões a respeito deste trabalho.

A. Ferramentas Usadas

Para a realização deste trabalho, foi escrito um programa em *Python3*. Foram utilizadas somente bibliotecas padrão da linguagem, como *sys* e *math*. Além disso, também foi feito o uso da plataforma *GitHub* para auxiliar na realização do trabalho de forma compartilhada.

II. Soluções dadas na implementação

A seguir, são descritas de forma teórica e prática as soluções propostas para cada uma das partes implementadas neste projeto.

A. Gerência de Processos

Na medida que processos chegam na CPU, a noção de um escolanador passa a ser necessária, do momento que de deve decidir qual processo vai entrar em uma CPU, dependendo de fatores como prioridade e ordem de chegada. Neste trabalho implementou-se o algoritmo *FIFO (First in, First out)*, *SJF (Shortest Job First)* e *Round Robin* (com quantum fixo igual a 2 unidades de tempo). Cada uma dessas metodologias tem sua vantagem e desvantagem teórica, podendo apresentar diferentes resultados em diferentes cenários. O algoritmo FIFO recebe aloca processos na CPU na medida que eles chegam, e cada processo fica na CPU até ele terminar. O algoritmo Shortest Job First vai procurar executar os processos do menos custoso ao mais custoso, dentre os processos que já chegaram na CPU. Já o Round Robin vai criar uma fila de processos, e o tempo alocado para cada processo será igual de acordo com o quantum. Quando um processo ficar esse determinado período na CPU, ele deverá sair para dar lugar a um novo processo.

A implementação em Python dessas formas de fazer a gestão dos processos foram feitas cada uma com um laço que sempre é igual a *'True'*, esperando uma condição de *'break'*, para cada algoritmo. Quando a *main.py* chama o módulo de Gerência de Processos, o arquivo de input é lido e cada algoritmo é executado em ordem. Cada laço tinha uma condição de parada, que testa se a fila de processos a serem executados já acabou. Cada algoritmo tem sua própria lista de processos, com cada entrada sendo o momento que o processo passou a poder ser executado, e o tempo que ele demora para terminar. Para todos os algoritmos, quando um processo termina de executar, ele sai da lista, e quando se identifica que a lista está vazia, a simulação termina. Existe uma diferença nessa

parte da implementação na parte do *Round Robin*, nesta parte foi mais fácil criar uma lista de processos que terminaram, e a lista de processos sendo executados é atualizada com o tempo que falta para a execução. Nos outros casos, o processo que entra na CPU para executar não sai e entra outra vez depois, por isso a diferença. Quando a lista de processos que terminaram no Round Robin tem o mesmo tamanho da lista de input, aí se detecta a condição de parada. O ato de ”executar o processo”ocorre no *FIFO* e no *SJF* por meio de um loop que faz um print que diz que o processo rodou. Sempre que um processo roda, o momento no qual a execução se encontra é atualizado de acordo com um contador, e o processo é retirado da lista de processos pendentes. No caso do *Round Robin* o mesmo loop existe, porém, os processos entram e saem várias vezes.

No *FIFO* a checagem para determinar se um processo pode executar é a mais simples dentre as implementações. Basicamente o programa recebe um arquivo com a ordem de chegada dos processos, e isso é colocado em uma lista. Para a primeira posição dessa lista, se checa se o momento de chegada é menor ou igual ao momento no qual o algoritmo está. Se sim, o processo toma posse da CPU, o tempo no qual o processo está é atualizado, o processo que rodou sai da lista de processos e pode-se checar o próximo candidato. Já o *SJF* apresenta a mesma estrutura, porém, no começo de cada iteração devemos passar por toda a lista de input, entre os processos que já chegaram na CPU. Na primeira iteração dessa busca, o primeiro processo que chegou passa a ser o candidato a ser executado, e depois disso, é necessário percorrer a lista de inputs atrás de um processo que fique menos tempo na CPU, caso exista. O *Round Robin* é o mais complicado, pois a lista de processos tem seus parâmetros atualizados, diferente dos outros algoritmos, no qual a lista de processos que faltam ser executados apenas diminui. Nesse caso, os processos rodam em uma fila que é criada por ordem de chegada. Cada processo fica duas unidades de tempo rodando e dá espaço para outro processo. Se faz a iteração do processo mais antigo ao mais novo da lista de inputs, caso o processo já tenha chegado na CPU, ele pode executar. No começo de cada iteração também se checa caso a CPU tenha que ficar ociosa, se não chegou nenhum processo para a execução. Quando um processo termina de rodar, ele entra na lista de processos terminados, quando o tamanho dessa lista é igual ao tamanho da lista de input original, o algoritmo acaba.

Ao final de cada laço um arquivo de texto com o resultado é atualizado com o tempo médio de *turnaround* (tempo de execução), tempo médio de resposta e o tempo médio de espera. Essas variáveis são atualizadas sempre que um processo é escalonado, comparando o tempo que o processo passou a ser executado com o tempo que ele estava pronto para ser executado, por exemplo. No final o módulo de Gerência de Processos retorna para a *main.py* um ”okay”e a execução do programa termina.

B. Gerência de Memória

O módulo de gerência de memória implementado tem como intuito simular a inserção e remoção de páginas da memória, com o uso dos algoritmos *FIFO*, *Second Chance* e *LRU*. Isso foi feito com a implementação de 4 funções: uma para cada algoritmo e uma função adicional responsável por chamar as funções que implementam os algoritmos.

O algoritmo FIFO trata a memória como uma fila, ou seja, as páginas são carregadas na memória em uma determianda ordem. Quando não há mais espaço disponível e é necessário carregr uma nova página, a primeira página a ter sido inserida (dentre as páginas carregadas em memória) é removida para que nova página possa ser inserida. Foi implementada a função fifo para simular este algoritmo. Esta função recebe como argumento a quantidade máxima de páginas comportada pela memória e uma lista que contém as páginas acessadas na ordem em que foram acessadas.

O algoritmo foi implementado de forma bastante simples: é mantida uma lista de inteiros que representa as páginas carregadas em memória. Caso exista espaço livre na lista e seja necessário carregar uma nova página, a nova página é adicionada no final da lista. Caso contrário, é removida a página da primeira posição da lista e a nova página é adicionada na última posição. Se a página desejada já está na lista, nenhuma ação é executada.

O algoritmo *Second Chance* consiste em manter um bit r para cada página carregada em memória. Quando uma página é iserida na memória, seu respectivo bit r tem o valor 1. Quando é necessário remover uma página da memória, são checadas as páginas carregadas a partir da página seguinte a última página substituída. Caso a página tenha o bit r com valor 1, o bit r tem seu valor modificado para 0 e a próxima página é verificada. Caso seja encontrada alguma página com o bit r com o valor 0, essa página é substituída. Esse procedimento é repetido para todas as páginas até que seja encontrada uma página com o bit r com o valor 0.

O algoritmo foi implementado com a função second_chance. Ela recebe como arguementos a quantidade máxima de páginas comportada pela memória e uma lista com as páginas acessadas na respectiva ordem em que foram acessadas. É mantida uma lista de inteiros que representa as páginas carregadass, uma lista que representa o bit r para cada página e uma variável que conta quantos acessos à memória foram feitos, pois a questão tinha uma exigência de zerar os bits r de todas as páginas a cada 3 acessos à memória. Quando uma página precisa ser carregada na memória, caso haja espaço livre, a página é inserida na lista, caso contrário, a lista de páginas percorridas é percorrida a partir da página marcada como à seguinte à última página substituída (página vítima) e o procedimento descrito no parágrafo anterior é aplicado. Caso a página desejada já esteja carregada, seu bit r tem o valor mudado para 1.

O algoritmo LRU consiste em monitorar quando cada página carregada em memória foi referenciada pela última vez. Quando é necessário substituir uma página, é substituída

aquela que foi referenciada pela última vez a mais tempo.

Este algoritmo foi implementado com o uso da função lru. Essa função recebe como argumento a quantidade máxima de páginas comportada pela memória e uma lista com as páginas acessadas na respectiva ordem em que foram acessadas. É mantida uma lista de inteiros que representa as páginas carregadas, uma variável (counter_value) que representa quando cada página foi referenciada pela última vez e uma lista (cpunter) para guardar para cada página o valor do counter_value. Quando uma nova página precisa ser inserida na memória e há espaço suficiente, página é inserida na lista e o (counter_value) é inserido no counter na mesma posição em que a página foi inserida na memória. Caso não haja espaço, a página com o menor (counter_value) associado é removida e a nova página é inserida em seu lugar. Caso a página desejada já esteja carregada, o valor de seu (counter_value) é atualizado.

C. Gerência de E/S

O módulo de gerência de entrada e saída implementado nesse programa possui a função de gerenciar a alocação do braço de leitura, que deverá percorrer os cilindros presentes no disco para atender a todas as requisições de acessos. Para isso, o módulo utiliza os seguintes algoritmos de escalonamento de disco: *FCFS (First Come, First Serve)* em que as requisições são atendidas de acordo com a ordem em que foram requisitadas, *SSF (Short Seek First)* em que o algoritmo atende sempre as requisições que demandam menor deslocamento do braço a partir de sua posição atual, e por último, *Scan* em que o braço começa de uma ponta do disco e se movimenta em direção a outra ponta atendendo a todos os requisitos assim que chega em cada cilindro específico.

A implementação desse módulo em Python foi realizada de forma bem simples e direta, o módulo possui 3 funções principais, uma para cada tipo de algoritmo de escalonamento de disco e uma função auxiliar bem simples chamada *"calcula_distancia"* que é chamada pelas funções *FCFS, SSF* e *Scan*. A função *"calcula_distancia"* recebe dois números inteiros como paramêtros de entrada e retorna como saida a distância entre esses dois números, como os cilindros são tratados como uma lista unidimensional, a distãncia é o módulo da subtração entre os parâmetros de entrada. Antes de realizar a chamada das funções mencionadas acima, o módulo lê uma série de números inteiros do arquivo referenciado pelo usuário sendo esses números: O número do último cilindro no disco, o cilindro sobre o qual a cabeça de leitura está inicialmente posicionada e a sequência de requisições de acesso. Os números lidos são então inseridos em diferentes listas que serão utilizadas por cada algoritmo citado, as listas que serão utilizadas pelas funções *SSF* e *SCAN* são ordenadas.

A função *FCFS* implementa o algoritimo de escalonamento de disco *FCFS*, a mesma chama a função *"calcula_distancia"* para calcular a distância entre o cilindro atual e o primeiro cilindro da lista de requisições de acesso, após isso, faz com que o cilindro da lista de requisições que acabou de ser atendido se torne o novo cilindro atual e chama a função *"calcula_distancia"* para calcular a distância entre o mesmo e o cilindro do próximo pedido de acesso da lista. Esse processo é repetito até que todos os pedidos da lista sejam atendidos. O resultado de cada iteração é somado na variável *Resultado_FCFS* cujo valor e mostrado ao usuário ao final da execução do algoritmo.

A função *SSF* implementa o algoritimo de escalonamento de disco *SSF*, para isso, utiliza uma lista *pedidos_SSF* que possui todos as requisições de acesso e o número do cilindro atual de forma ordenada. Como a lista já está ordenada, a função apenas usa o index (posição na lista) do cilindro atual para calcular a distância entre o valor apontado pela posição index-1 e index+1 da lista para encontrar a menor distãncia entre o cilindro atual e o cilindro de uma requisição de acesso adjacente. O valor da distância é armazenado em uma váriavel *resultado_SSF* e a posição da lista onde se encontrava o cilindro com a requisição de acesso, que demandou o menor descolamento do braço de leitura, se torna o novo index do cilindro atual. O processo é repetido até que todas as requisições de acesso sejam atendidas e para cada iteração, o valor da menor distância calculada é somada a variável *resultado_SSF*. O valor da variável é mostrado ao usuário ao final da execução do algoritmo.

Por último, A função *Scan* implementa o algoritimo de escalonamento de disco *Scan*, utilizando uma lista *pedidos_Scan* que possui todas as requisições de acesso, o número do cilindro atual e os limites das pontas dos cilindros (0 e último cilindro) de forma ordenada, além de uma varíavel *direcao_esquerda_direita que funciona como uma flag*, indicando a direção que o braço de leitura está seguindo. Em listas que possuem números de cilindros repetidos, o algoritmo utiliza um *while* para garantir que o index (posição na lista) do cilindro atual referencie sempre a última ocorrência da repetição. Quando a flag está setada como *True*, o algoritmo calcula a distância entre o valor da lista na posição index, que inicialmente referencia o cilindro atual, e o valor encontrado na posição index-1 armazenando o resultado encontrado na variável *Resultado_Scan*. Após isso, o index irá apontar para a posição index-1 e será realizado o calculo entre o valor apontado pelo novo index e index-1, o processo irá se repetir até que index-1 aponte para o valor do cilindro 0, ou seja, até encontrar a ponta da lista. A partir desse ponto, a flag será setada como *False* e o algoritmo irá funcionar de forma análoga mas para o outro sentido, calculando as distâncias entre os valores apontados por index e index+1, para cada iteração o valor calculado será somado a variável *Resultado_Scan* e após atender todos as requisições de acesso o valor de *Resultado_Scan* é apresentado ao usuário.

III. Principais dificuldades encontradas e soluções

Não foram encontradas dificuldades, exceto para a compreensão do algortimo *Second Chance*. Essa dificuldade foi sanada com uma pesquisa na internet que mostrou um vídeo, [1], que explica o algoritmo de forma detalhada. Este vídeo pode ser encontrado na seção de Referências.

IV. Papel de cada aluno na realização do trabalho

O autor Guilherme Braga focou na parte de gerenciamento de processos, o autor Gustavo Carvalho realizou a implementação do módulo de gerenciamento de memória, o autor Gabriel Matheus trabalhou na parte de gerência de E/S e a autora Ana Paula ajudo no gerenciador de processos e na revisão dos algoritmos, com a realização de testes.

V. Conclusões

Desta forma, pode ser constatado que a linguagem Python foi o suficiente para fazer simulações simples de como que um sistema operacional lida com requisições e como lida com diversos recursos em seu sistema.

Referências

[1] *(1) Second Chance Algorithm - YouTube*. https://www.youtube.com/watch?v=C26qsPwf-Js. (Accessed on 05/13/2021).