Relatorio do Tabalho Prático:
Implementação do pseudo-SO
Grupo 10

Felipe Fontenele dos Santos, 19/0027622
João Pedro Sadéri da Silva, 17/0126021
Marcus Vinicius Oliveira de Abrantes, 19/0034084

[1]Universidade de Brasília (UnB)
Fundamentos de Sistemas Operacionais

1. Introdução

Neste relatório trataremos da implementação do pseudo-SO desenvolvido pelo grupo 10 da disciplina de Fundamentos de Sistemas Operacionais. Ao longo deste documento, iremos expor a linguagem usada, informações teóricas e práticas acerca da implementação, dificuldades e soluções.

Em cada seção discutiremos quais foram as inspirações teóricas para as abstrações utilizadas para dar ao projeto funcionalidade simplificada de um Sistema Operacional. Será enunciado as soluções e os seus passos e o processo para desenvolvimento da versão final.

1.1. Linguagem e bibliotecas utilizadas

O trabalho foi desenvolvido na linguagem **Python**. Todas as bibliotecas utilizadas são naturais da linguagem e não requerem a instalação de nenhum componente externo.

2. Implementação

Para a realização do trabalho em grupo, o projeto foi divido em módulos , de forma que cada integrante pudesse trabalhar de forma independente em uma parte do pseudo-SO, simplificadas de um SO real para atender apenas às necessidades do projeto. À medida que cada integrante finalizou a parte por qual estava responsável, os módulos foram unidos e integrados para possuírem funcionamento único.

Ao todo o projeto é constituído de 6 módulos principais:

- **Um módulo auxiliar**, responsável por fazer o **parsing** dos arquivos de textos fornecidos e convertê-los em dados que pudessem ser tratados dentro da implementação. Este modulo foi desenvolvido e trabalhado em conjunto pelo grupo todo;
- **Um módulo de processos** [João], responsável por fazer a instanciação de cada processo, além de realizar o escalonamento destes;
- **Um módulo de memória** [Marcus], abstração que representa a persistência de dados na memória.
- **Um módulo de Entrada/Saída** [Marcus], da mesma forma, uma abstração para a manutenção de componentes externos ao SO.
- **Um módulo de Arquivos** [Felipe], atende à funcionalidade de criação e deleção de arquivos em memória, além dos erros advindos de operações indevidas.
- **Um módulo de gerenciamento de recursos** [João], de forma semelhante ao processador, integra e executa os comportamentos de cada módulo.

2.1. Gerência de Memória

O módulo de gerência de memória abstrai o comportamento da memória RAM. Dada as especificações do projeto, este módulo não armazena os conteúdos de processos e arquivos, mas ao em vez disso, simula a alocação na memória dos recursos que são necessários para cada módulo. A implementação se encontra na pasta **memoria**.

Na gerência de memória, deve-se implementar um conjunto de blocos em que os dados são armazenados de forma a contínua. A memória possui ao todo 1024 mb sendo que 64mb do total estão reservadas para processos de tempo real. Cada bloco possui 1mb.

Dado que não haveria a necessidade de armazenar as informações para posterior leitura, e ao mesmo tempo, que a linguagem python aloca as estruturas de dados de forma dinâmica, optamos por unir o mapeamento de blocos livres e ocupados com a estrutura de armazenamento no formato de uma lista encadeada.

Desta forma, cada nó da lista possuirá as seguintes informações:

```
class Node:
    def __init__(self, id, tipo, tamanho):
        self.id = id
        self.tipo = tipo
        self.tamanho = tamanho
        self.next = None
```

Com id sendo fornecido pelo processo, para fazer referência à localização na memória, o tamanho total ocupado pelos dados, e o tipo, definido no *Enum*:

```
class TipoMemoria(Enum):
    PRCS_LIVRE = 1
    PRCS= 2
    TREAL_LIVRE = 3
    TREAL= 4
```

Desta forma, caso um bloco na área reservada a tempo real esteja livre, possuirá tipo **TREAL_LIVRE**, e de forma equivalente no restante da memória **PRCS_LIVRE**. Ao ser ocupado por um dado, o node terá respectivamente **TREAL** e **PRCS**, possuindo o *id* e o tamanho especificado. A gerência de memória abstrai a localização específica do dado, que pode mudar com a utilização de outras funções, mas é possível obter sua posição somando o tamanho de cada nó da lista percorrido até que se encontre o desejado.

Depois foram implementadas funções de compactação e coalescência. Dado que cada nó possui um valor referente a seu tipo, basta ordena-la de forma decrescente e por conseguinte as regiões ocupadas por dados estarão concentrados nos primeiros endereços de cada região reservada. Já para coalescer, percorre-se as listas e compara-se o tipo de nós consecutivos, caso sejam equivalentes ao tipo livre de sua região, basta transformá-los em um nó com a soma dos tamanhos de cada.

2.2. Gerência de Processos

A gerencia de processos apresenta uma abstração de uma CPU que deve dar prioridade a dois tipos de processos, processos de tempo real e processos do usuário. Após ser definido um tipo de elemento chamado processo, que armazena todas as informações

referentes ao processo, a lista de todos os processos é dividida em duas novas filas. A primeira delas se refere aos processos de tempo real e a segunda tem referência aos processos de usuário.

Durante a execução do método run, presente na classe Gerenciador_de_processos, verifica-se o tempo de inicialização de um processo é condizente com o tempo atual e nesse caso, atualiza a lista de processos pronto de tempo real. Após isso, basta reordenar a lista pela prioridade contida em cada processo. Como foi utilizado uma estrutura de lista baseando-se em uma fila, a ordem de execução é baseada também na ordem de chegada, mantendo assim uma fila FIFO para os processos de tempo real.

Para a fila de processos do usuário, optamos em utilizar uma grande fila que pode trabalhar com prioridades entre 1 até 10000 para os processos de usuário. Quando cada processo entra na fila seguindo com base na prioridade inicial aplicada a ele. Essa fila então é ordenada com base na prioridade entre os processos. Como a ordem de chegada também é considerada, processos que apresentam a mesma prioridade tem o seu fator de desempate relacionado ao momento de entrada na fila.

Para evitar *starvation* em relação a processos de usuário com prioridade baixa em relação aos outros, após cada quantum de execução de um processo de usuário, o gerenciador de processos aumenta gradativamente o valor da prioridade desse processo, fazendo com que processos com 20 operações sejam realizados em 4 blocos de 5 operações, podendo ser intercalados com outros processos de prioridade inferior ao atual, garantindo que um processo de prioridade baixa não fique esperando para executar durante muito tempo em relação aos demais. Para decidir a ordem de execução, verifica-se a existência de processos de tempo real sempre, caso afirmativo, procura-se executar esse e caso contrário, dá posse da pseudo-cpu à um processo de usuário.

2.3. Gerência de Entrada/Saída

Neste módulo deve-se implementar uma interface que simule a interação entre S.O. e dispositivos externos. Em um sistema real, cada dispositivo possuíria um *driver* que executará os comandos próprios do dispositivo, e que é disparado por *syscalls* pelo S.O.

Na implementação deste projeto, para realizar a simulação simplificada de um *device driver*, todos os dispositivos herdam da classe abstrata **driverES** e precisam implementar os seus métodos, com os quais o S.O. poderá interagir. Estes seguem um padrão, em que cada dispositivo fica responsável for definir sua função específica.

Na classe **GerenciadorIO** é feito o mapeamento do *id* gerado de cada dispositivo com seu respectivo módulo.

```
class GerenciadorIO:
    def __init__(self) -> None:
        self.IOList = {}
```

Como no escopo deste projeto, cada sistema I.O. não deve realizar funções complexas, seu comportamento é iniciado ao disparar a função **executar()** definida como padrão dos que herdam a classe abstrata de **driverES**.

2.4. Gerência de Arquivos

A gerência de arquivos visa receber informações da E/S com o conteúdo a ser gravado de uma maneira especifica na memória. Este foi um componente onde sentimos uma certa dificuldade de conectá-lo aos outros módulos e, por isso, não conseguimos realizar a validação dos processos existentes e da autorização dos processos que podiam ou não excluir o arquivo presente na memória.

Ele funciona de maneira simples, realizando o first-fit no momento da inserção da memória, ou seja, a primeira sequência de blocos livres que satisfaçam o tamanho do arquivo ele irá utilizá-la.

3. Conclusão

Neste trabalho, tivemos a oportunidade de aprofundar os conhecimentos estudados em sala de aula. Ao investigar a implementação de cada parte, fomos desafio a entender um SO funciona na forma prática. Embora o projeto desenvolvido seja apenas um abstração de um sistema operacional real, dado que este demandaria um esforço imensamente maior, foi possível compreender de forma lógica quais são as decisões que devem ser tomadas e os gerenciamentos que acontecem em um sistema real.

Assim concluímos que o desenvolvimento deste projeto foi fundamental para esclarecer algumas das dúvidas que surgiram em sala de aula. Além de revelar alguns dos desafios de programar um sistema operacional real.