Especificação do trabalho prático de FSO

Antônio Vinicius de Moura - 190084502
Luca Delpino Barbabella - 180125559
Marcelo Aiache Postiglione - 180126652

Dep. Ciência da Computação - Universidade de Brasília (UnB)
Fundamentos de sistemas operacionais

1. Descrição das ferramentas/linguagens usadas

1.1. **Python 3.10:** A mais recente versão do Python, uma das linguagens de programação mais populares. Ele fornece recursos aprimorados para ajudar a criar aplicativos mais eficientes e poderosos. Esta versão inclui melhorias no desempenho, novos recursos, melhorias na sintaxe e diversos outros aprimoramentos.

1.2. **Git / Github:** Git é um sistema de controle de versão que permite que os usuários mantenham o controle de modificações em um arquivo ou conjunto de arquivos. O Github, por sua vez, é uma plataforma que oferece serviços de armazenamento de código-fonte e gerenciamento de versões usando o sistema Git.

1.3. **Excalidraw:** Ferramenta de design de desenho colaborativo em tempo real projetada para ajudar a criar ilustrações e escrever ideias.

1.4. **Visual Studio Code:** Editor de código-fonte desenvolvido pela Microsoft para Windows, Linux e macOS. Ele suporta a maioria das principais linguagens de programação. O Visual Studio Code é equipado com diversas ferramentas para ajudar os desenvolvedores a criarem código de melhor qualidade.

1.5. **Discord:** Aplicativo de bate-papo em grupo gratuito que permite que usuários criem e usem servidores para comunicações por voz, texto e vídeo. O Discord fornece recursos como salas de bate-papo, salas de voz, enviar mensagens diretas e muito mais.

2. Descrição teórica e prática da solução dada

2.1. Módulo de Processos/Módulo de Filas

Os módulos de processos e de filas foram implementados na classe *ProcessManager* e na classe *Process*. As filas de prioridade são representadas por listas (que são armazenadas na classe ProcessManager), enquanto os processos são representados por objetos (essas classes são do tipo process).

Inicialmente, é executado o método *process_preemption()*, onde toda a lógica de escalonamento de processos é feita. O escalonamento de processos funciona da seguinte maneira:

1. Se há função na CPU, aumentamos o tracker de tempo de execução daquele processo. Se o processo executou todas as suas instruções, o remove da CPU (verificado com o método *check_process_finish()*).

2. Tenta colocar na CPU o processo disponível de maior prioridade (se não houver nenhum processo, simplesmente deixa passar).
3. Impede que aconteça preempção caso o processo corrente seja um processo de tempo real e não tenha acabado.
4. Checa se há um processo de prioridade maior que o atual (utilizando o método *check_higher_priority()*), e se houver, o carrega (utilizando o método *load_process()*).
5. Se não houver processo de prioridade maior, verifica se há mais processos de prioridade igual ao do que está na CPU. Se o processo não tiver terminado e existirem mais processos de mesma prioridade, rotaciona a fila (usando o método *process_queue_rotation()*).
6. Se não existirem mais processos de mesma prioridade do que o carregado na CPU e ele tiver sido finalizado, verificar fila de prioridade menor (usando o método *check_lower_priority()*) e carrega na CPU caso alguma delas tiver processos.

Em seguida, chamamos o método *age_process()* para fazer o aging de todos os processos de prioridade 2 e 3. Não fazemos o aging dos processos de prioridade 1 e 0, porque processos de prioridade 0 são tipos especiais de processos (um processo com prioridade 1 não pode passar a ter prioridade 0), e porque a prioridade 0 já é a máxima possível.

2.2. Módulo de Memória

O módulo de memória está implementado na classe *MemoryManager*. A memória é representada por um array inicializado no construtor da classe com todos os elementos iguais a zero e seu tamanho é definido pelas constantes *REAL_TIME* e *USER*, que definem a quantidade de blocos reservados para processos de tempo real e de usuário e possuem valores de 64 e 960, respectivamente.

Para a alocação de um processo, a função *allocate* é utilizada. Ela recebe como parâmetro um objeto da classe processo que requer uma alocação e realiza uma varredura pela memória buscando a quantidade de blocos contíguos exigidos, isso é verificado consultando o campo *memory_blocks* do objeto. A prioridade do processo é verificada para que a varredura ocorra na área correta da memória. A função itera pela memória contando os blocos contíguos livres e aloca o processo no primeiro intervalo de tamanho igual ao campo *memory_blocks* encontrado. Após identificar um intervalo a função atualiza o array que representa a memória indicando os novos blocos ocupados e retorna o valor *True*. Caso não seja encontrado um intervalo com tamanho adequado, o retorno será *False* e o processo não será alocado. Uma pequena otimização com o intuito de evitar uma varredura desnecessária por todos os blocos da memória foi realizada. Caso um bloco livre tenha sido encontrado é verificado se existe a possibilidade de encontrar um intervalo até o fim da

memória, ou seja é verificado se o índice atual na memória até o índice final consegue comportar a quantidade de blocos exigida pelo processo.

Os processos são desalocados utilizando a função *free*, que recebe o objeto processo, verifica os campos *first_block* e *memory_blocks* e libera os blocos indicados da memória.

2.3. Módulo de Recurso

O módulo de recurso está implementado na classe *ResourceManager*. Essa classe é instanciada passando 1 scanner, 2 impressoras, 1 modem, 2 dispositivos SATA e um Lock implementado com a biblioteca *multiprocessing* que é utilizado para alocação e liberação dos dispositivos de E/S.

Existem 3 funções essenciais para o correto funcionamento do módulo de recursos, são elas:

***verify_allocated_resources()*:** Esta classe é responsável por verificar se o processo já possui os recursos necessários para ser executado. Primeiro checa se processo precisa de qualquer recurso, caso afirmativo verifica se já possui e retorna.

try_get_resource(): O objetivo desta classe é ajudar a garantir que os recursos estejam disponíveis para os processos que precisam deles. Ela verifica se os recursos que o processo precisa estão disponíveis, caso estejam ela aloca esses recursos.

allocate_resources(): Esta classe foi criada para gerenciar os recursos. Primeiramente, verifica se o processo é de tempo real, caso afirmativo retorna que não é possível executar o processo. Após isso, chama a classe que verifica se os recursos necessários estão disponíveis para um determinado processo ou a classe que aloca esses recursos.
No final, é retornado se o processo pode ou não ser executado.

free_resources(): Esta função é responsável por liberar os recursos utilizados pelo processo. A função verifica se o processo usou os recursos necessários, se usou, atribui o valor -1 para esses recursos, indicando que eles estão disponíveis para uso.

As funções *allocate_resources*() e *free_resources*() utilizam a variável "Lock" para entrar na região crítica do código, o que garante que apenas um processo esteja acessando os recursos de cada vez.

2.4. Módulo de Arquivos

O sistema de arquivos está implementado na classe *FileManager*. A classe *File* representa um arquivo. Na leitura do arquivo de texto, inicializamos o sistema de arquivos e para cada arquivo descrito um objeto *File* é criado e adicionado ao dicionário *files* da classe *FileManager* e também é feita a atualização do mapa do disco, um array que representa os blocos físicos. As operações são salvas em um objeto da classe *ProcessOperation* e

colocadas em na lista *FileManager.operations.* O disco é inicializado com o tamanho lido e com os blocos dos arquivos sinalizados como ocupados.

Para criar arquivos, a função recebe as informações do arquivo e a id do processo. Realiza uma varredura e usando o método *first-fit* aloca o arquivo ou não dependendo da disposição dos blocos livres no momento da solicitação.

A função de deletar um arquivo checa o tipo do processo para verificar suas permissões, caso o processo tenha as devidas permissões o arquivo é removido de *FileManager.files* e os blocos do disco são marcados como livres.

A função *operate_process* realiza uma operação do processo recebido como parâmetro e a retira da lista de operações pendentes.

As mensagens retornadas por essas funções são salvas no log da classe *FileManager* e posteriormente mostradas na tela ao final da execução de todas as tarefas.

Para mostrar o mapa de ocupação ao final da execução a string que representa o disco tem o seguinte formato inicial "| | | |...| |", então podemos mapear o índice do bloco ocupado para o índice da string usando a fórmula 2 * (n + 1) - 1, onde n é o bloco ocupado começando em 0. Assim passamos pelos blocos ocupados pelo arquivo e mudamos o espaço vazio da string com o nome do arquivo que ocupa aquele bloco. Ao final teremos o nome do arquivo, de uma letra, no índice de seus blocos na string.

3. Descrição das principais dificuldades encontradas durante a implementação

- Dificuldade para pensar em como salvar as operações descritas no arquivo de texto

4. Soluções utilizadas para as dificuldades encontradas

4.1. Dificuldade para pensar em como salvar as operações descritas no arquivo de texto:

Criamos uma classe que representa cada operação e salvamos todas as informações necessárias. Os objetos são adicionados à lista de operações do file manager e quando um processo está na CPU, nós verificamos se existe alguma operação pendente desse processo, realizamos a operação e a retiramos da lista.

5. Papel/função de cada aluno

Marcelo: Módulo de memória, módulo de arquivos e integração dos módulos de memória, processos e arquivos no programa principal (main).
Antônio: Módulo de arquivos, módulo de recursos, integração dos módulos de arquivos e recursos no programa principal (main).

Luca: Módulo de recursos, módulo de processos, integração dos módulos de recursos e processos no programa principal (main).

6. Repositório Github

Para acessar todos os códigos utilizados no projeto, basta clicar no link: https://github.com/marcelo3101/pseudo-so.