Relatório de Implementação do PseudoSO

Nicole de Oliveira Sena 190114860@aluno.unb.br
Filipe de Sousa Fernandes 202065879@aluno.unb.br
Victor Santos Candeira 170157636@aluno.unb.br

Universidade de Brasília, 14 de julho de 2025

Resumo

1. Introdução

O presente relatório descreve o desenvolvimento de um projeto acadêmico que simula o funcionamento básico de um sistema operacional, chamado de **pseudo SO**. O objetivo principal do projeto é replicar, de forma mais simplificada, os principais componentes e funcionalidades de um sistema operacional real, tais como gerenciamento de processos, controle de memória, manipulação de dispositivos de entrada e saída, além de um sistema de arquivos básico.

O pseudo SO foi desenvolvido com o propósito de consolidar os conhecimentos teóricos abordados na disciplina de **Fundamentos de Sistemas Operacionais**, permitindo que os alunos explorem como os subsistemas de um sistema operacional interagem entre si.

O projeto é organizado em módulos específicos. O núcleo (`core/`) engloba os principais mecanismos do pseudo SO: criação e escalonamento de processos, gerenciamento de memória e recursos, e controle de arquivos. Um módulo auxiliar (`utils/`) é responsável por fazer o parsing dos arquivos de entrada que alimentam o sistema com dados de processos e operações. Também foi incluída uma suíte de testes (`tests/`) para garantir a integridade e o bom funcionamento dos componentes. O sistema é executado a partir do arquivo `main.py`, que atua como despachante, coordenando a execução dos processos simulados.

Este relatório apresenta a arquitetura, os componentes implementados, os desafios enfrentados durante o desenvolvimento e os testes realizados para validação do projeto.

2. Arquitetura Geral do Sistema

O projeto do PseudoSO foi estruturado de forma modular, visando favorecer uma implementação em grupo. Cada módulo foi implementado com responsabilidades bem definidas, o que facilita a organização, manutenção e testes do código. A função de cada diretório é detalhada a seguir:

- **main.py:** Ponto de entrada do sistema. Atua como despachante, controlando o fluxo da simulação e acionando os componentes centrais conforme o andamento da execução.
- **core/:** Contém os módulos fundamentais.
  - **processo.py:** Define a estrutura e o comportamento dos processos simulados.
  - **filas.py:** Gerencia as filas de escalonamento (pronto, bloqueado, finalizado, etc).
  - **memoria.py:** Implementa a lógica de alocação e liberação de memória.
  - **recursos.py:** Controla o uso de dispositivos de I/O.
  - **arquivos.py:** Simula um sistema de arquivos simples com operações básicas.
  - **sistema_operacional.py:** Coordena a interação entre todos os módulos do núcleo.
- **utils/:** Contém o módulo auxiliar `parser.py`, responsável por ler e interpretar os arquivos de entrada com a descrição dos processos e operações em arquivos.
- **tests/:** Diretório com testes unitários e de integração que validam o funcionamento de cada componente do sistema.
- **validation_files/:** Armazena os arquivos de entrada utilizados na simulação. Os arquivos `process*.txt` descrevem os processos e seus atributos; os `files*.txt` contêm comandos relacionados ao sistema de arquivos.
- **read_me.md:** Documento com instruções básicas de uso e estrutura do projeto.
- **relatorio.pdf:** Documento que descreve o desenvolvimento do projeto (este relatório).

Essa arquitetura modular facilita o desenvolvimento incremental, além de ajudar no rastreamento de bugs. Outro benefício dessa arquitetura é a execução de testes independentes para cada módulo do sistema.

3. Componentes Principais (Core)

3.1 Processo

O módulo `processo.py` define a classe **Processo**, que configura a unidade básica de execução dentro do nosso PseudoSO. Cada instância dessa classe encapsula as informações e os recursos cruciais para simular um processo em execução, assim como seu ciclo de vida durante a simulação.

Os processos são inicializados com os seguintes atributos: **pid** (identificador único do processo), **chegada** (momento em ciclos de clock para admissão), **prioridade** (nível de prioridade), **tempo_cpu** (tempo total de CPU necessário), **blocos_mem** (número de blocos de memória requisitados), **impressora**, **scanner**, **modem**, **sata** (quantidade de dispositivos específicos requisitados), **offset** (endereço base da alocação de memória), **executado** (contador de vezes escalonado), **instrucoes_executadas** (número total de instruções simuladas), e **lista_id_operacoes** (lista para associar o processo a operações no sistema de arquivos).

O método **executar_processo(autorizacao_cpu, printar=False)** simula a execução do processo por um tempo definido por `autorizacao_cpu`, atualizando instruções executadas e tempo de CPU restante, e opcionalmente imprime o andamento da execução. O método **print_execucao_processo(autorizacao_cpu)** imprime no console o estado atual do processo, distinguindo entre início (STARTED) e retomada (RESUMED), exibindo o progresso e indicando quando o processo finaliza seu tempo de CPU (emitindo "SIGINT").

3.2 Filas

O módulo `filas.py` implementa a classe **Escalonador**, dedicada ao gerenciamento das filas de processos prontos e à política de escalonamento do PseudoSO. O sistema usa uma abordagem híbrida com escalonamento prioritário multinível com aging para processos de usuário e FIFO sem preempção para processos de tempo real.

- Existem quatro filas principais:
  - **fila_tempo_real**: para processos prioridade 0 (tempo real), executados até o fim sem preempção, política FIFO;
  - **fila_usr_p1, fila_usr_p2, fila_usr_p3**: filas para processos de usuário com prioridades 1, 2 e 3, escalonados por quantum fixo com aging.
- Essas filas são agrupadas em:
  - **filas_usr**: lista contendo as três filas de usuário;
  - **fila_pronto**: estrutura geral contendo a fila de tempo real seguida das filas de usuário.
- E os **parâmetros de escalonamento** são:
  - **quantum = 1**: tempo máximo de CPU para processos de usuário por iteração;
  - **aging = 5**: número máximo de execuções sem progresso antes de aumentar prioridade;
  - **max = 1000**: máximo de processos simultâneos nas filas.

O método **adicionar_processo(processo)** adiciona o processo na fila adequada por prioridade; descarta se limite máximo atingido. Já o método **proximo_processo()** seleciona e remove próximo processo para execução, priorizando processos de tempo real. O **tempo_autorizado(processo)** determina tempo de CPU concedido (quantum para usuário, tempo total para tempo real). Por fim, **aplicar_aging(processo)** aplica aging para evitar inanição, promovendo processos que ficaram muito tempo nas filas.

3.3 Memória

O módulo `memoria.py` implementa a classe **GerenciadorMemoria**, que simula alocação e liberação de memória como um vetor de blocos contíguos, sem paginação nem memória virtual.

A memória possui 1024 blocos:

- **Blocos 0 a 63**: reservados para processos de tempo real.
- **Blocos 64 a 1023**: para processos de usuário.

A memória é modelada como uma lista onde cada posição guarda o PID do processo ocupante ou -1 se livre. A alocação deve ser contígua e ocorre nas áreas específicas conforme a prioridade do processo.

O método **alocar(processo)** reserva os blocos necessários, atualiza o atributo `offset` do processo e marca os blocos com o PID. O método privado **_encontrar_bloco_contiguo(inicio, fim, tamanho)** busca espaço contíguo suficiente. O método **liberar(processo)** libera os blocos ocupados e reseta o `offset`. O método **print_mapa_ocupacao()** exibe o estado atual da memória no console.

3.4 Recursos (E/S)

O módulo `recursos.py` define a classe **GerenciadorRecursos**, que gerencia a alocação e liberação de dispositivos de entrada e saída, garantindo que não haja conflitos no uso simultâneo.

Dispositivos disponíveis:

- **Scanner**: 1 unidade
- **Impressoras**: 2 unidades.
- **Modem**: 1 unidade.
- **SATA (discos)**: 2 unidades.

Cada recurso armazena o PID do processo ocupante ou -1 se livre. O método principal **alocar(processo)** tenta reservar todos os recursos solicitados pelo processo, usando `_alocar()` para cada recurso individual. Em caso de falha, chama **_rollback_alocacao()** para liberar recursos já alocados. O método **liberar(processo)** devolve os recursos usados por um processo, verificando a propriedade para evitar conflitos. O método **print_aloc_recursos()** mostra o status atual dos recursos.

3.5 Arquivos

O módulo `arquivos.py` implementa a classe **GerenciadorArquivos**, que simula um sistema de arquivos com alocação contígua inspirado em FAT.

- **mapa_ocupacao**: lista representando o estado do disco, com caracteres para arquivos ou espaço para blocos livres.
- **num_blocos**: número total de blocos, definido dinamicamente.
- **operacoes**: lista de operações de arquivos agendadas.
- **livre**: caractere padrão para blocos livres (' ').

O método **iniciar_filesystem(operacoes_arquivos)** recebe dados iniciais, preenche o mapa de ocupação e prepara as operações, cada uma com um identificador único. O método **aplicar_operacao(id_op)** executa uma operação ainda não realizada,

identificada por id_op. As operações podem ser **cod_operacao == 0** (criação de arquivo) e **cod_operacao == 1**: (deleção de arquivo).

A operação é marcada como **executada**, e seu sucesso é determinado pelo resultado das funções `criar_arquivo` ou `deletar_arquivo`. A **criar_arquivo(index)** percorre o disco procurando blocos contíguos livres suficientes. Se encontrar, marca os blocos com o nome do arquivo e atualiza a operação como **bem-sucedida**. Caso contrário, a criação falha (*falta de espaço contíguo*). Já a **deletar_arquivo(index)** percorre o disco em busca de blocos ocupados pelo arquivo alvo. Se encontrado, libera todos os blocos ocupados e marca a operação como **concluída com sucesso**. Se o arquivo não existir, a deleção **falha**.

A função **identificadores_ops_de_processo(pid)** retorna a lista de identificadores de operações associadas a um dado processo. A **print_mapa_ocupacao()** imprime o estado atual do disco e a **print_resultado_operacoes()** gera um relatório detalhado com o resultado de todas as operações executadas, indicando **sucesso ou falha com justificativa**.

3.6 Sistema Operacional

O módulo `sistema_operacional.py` implementa a classe **SistemaOperacional**, que orquestra a execução dos componentes do PseudoSO, atuando como despachante.

Principais atributos:

- **escalonador**: instância de `Escalonador`.
- **memoria**: instância de `GerenciadorMemoria`.
- **recursos**: instância de `GerenciadorRecursos`.
- **arquivos**: instância de `GerenciadorArquivos`.
- **processos**: lista de processos a despachar.
- **operacoes_arquivos**: operações associadas aos processos.
- **executando**: processo atualmente em execução.
- **tempo**: contador do tempo do sistema.

O método `executar()` simula um ciclo contínuo de operação do sistema, onde a cada unidade de tempo o sistema executa a etapa de **Inserção de Novos Processos** processos com `chegada` menor ou igual ao tempo atual são considerados. Para cada um associam-se as operações de arquivos correspondentes e tenta-se alocar memória. Se houver sucesso e espaço na fila de prontos, o processo é inserido, senão, o processo volta à lista de espera. Depois é a **Escolha do Próximo Processo** onde o escalonador fornece o próximo processo pronto a ser executado com base na prioridade e no algoritmo FIFO/quantum.

Na **Execução do Processo** se houver processo a ser executado tenta-se alocar os recursos de E/S, o processo recebe um tempo de CPU (quantum ou o tempo restante) eo método **executar_processo()** simula a execução do processo.Caso ele ainda tenha tempo restante, é reencaminhado à fila de prontos (com aging aplicado, se necessário).Senão as operações de arquivos associadas são executadas e a memória é liberada. Os recursos de E/S são sempre liberados ao final da execução. No **Término da Simulação** o loop se encerra quando não há mais processos a serem carregados nem processos em execução nas filas.

Por fim ao término, imprime-se o tempo final da simulação, resultado das operações de arquivos e estado final do disco (mapa de ocupação). A função auxíliar **msg_processo_criado()** exibe detalhes do processo criado, como prioridade, tempo de CPU, recursos solicitados e bloco de memória alocado.

4. Entrada de Dados

O projeto utiliza dois arquivos .txt como fonte de entrada: um contendo os processos a serem executados e outro com as operações do sistema de arquivos. Esses arquivos são armazenados na pasta `validation_files/` e seguem um formato padronizado para facilitar a leitura e interpretação automática.

A leitura e interpretação desses arquivos são responsabilidade do módulo `parser.py`, da pasta `utils/`. Esse módulo implementa funções para ler e validar os arquivos de entrada, e depois transformá-lo em objetos Python que o sistema possa manipular.

5. Testes

Para garantir a confiabilidade do PseudoSO, foram desenvolvidos testes específicos para cada módulo, que estão na pasta `tests/`. Foi adotado uma abordagem de testes unitários para cada módulo e um teste de integração, para focar tanto na verificação isolada de funcionalidades quanto no comportamento em conjunto dos módulos.

6. Dificuldades e Limitações

Durante o desenvolvimento do projeto do **pseudoSO**, enfrentamos diversas **dificuldades técnicas e conceituais** que impactaram o design e a implementação.

Uma das primeiras questões foi a necessidade de implementar **validações e tratamentos específicos para os códigos dos dispositivos de entrada e saída**. Conforme o Filipe mencionou, era importante tratar corretamente situações em que a **solicitação de recurso fosse inválida** para evitar inconsistências e travamentos no sistema.

Outra dificuldade está relacionada ao **ciclo de vida dos processos**. Inicialmente, processos que não conseguiam alocar recursos ou memória eram simplesmente retornados para a fila de prontos, o que poderia gerar **ciclos infinitos de espera**. Após discussão, definiu-se que o procedimento correto seria **matar o processo nesses casos**, tratando-o como encerrado por erro e evitando loops desnecessários.

Além disso, o fato de se tratar de um **pseudo sistema operacional** gerou complexidades que não existem em sistemas reais. Por exemplo, a execução de operações de arquivo por processos que, na simulação, não existem de fato. Em um SO real, operações de arquivos são parte das instruções do processo; aqui, foi preciso **armazenar essas operações em uma estrutura separada dentro do módulo de sistema de arquivos** — uma solução artificial, mas necessária para manter a coerência do sistema simulado.

Outro ponto ambíguo foi a **limitação da fila de prontos**, que aceita até 1000 processos. O requisito não deixa claro o que fazer quando esse limite é atingido. A solução implementada no `dispatcher` foi **devolver o processo excedente para uma estrutura de pré-prontos**, criando algo como um "quarto estado" no ciclo de vida dos processos, uma etapa de "solicitando entrada". Embora isso não reflita exatamente o funcionamento de um SO real, faz sentido no contexto da simulação, que poderia armazenar temporariamente processos em disco enquanto a fila principal se esvazia. Ficou em dúvida se essa **limitação da fila de prontos é uma prática usual em sistemas operacionais reais**.

Uma dificuldade adicional está na **definição pouco clara sobre o que seria uma instrução, um ciclo de clock e o quantum do**

processo, definido como 1 ms. Pelos exemplos e saídas do sistema, interpretamos que esses conceitos são equivalentes na prática, simulando um processador de núcleo único rodando a 1 kHz. Essa abordagem, apesar de simplificada e pouco usual, permitiu que o sistema fosse executado corretamente e as saídas conferissem com as esperadas. O `dispatcher`, nesse modelo, executa um loop com um contador representando microsegundos. Para processos de tempo real, o tempo de execução é consumido integralmente em um único passo, pois esses processos não são preemptivos e não há necessidade de ficar verificando a fila de prontos.

Nesse cenário simplificado, a **concorrência no uso dos recursos não é um problema**. Cada processo tenta adquirir seus recursos quando assume a CPU e os libera assim que a solta, pois não há sentido em manter recursos ocupados sem saber quando o processo será retomado.

Essas dificuldades evidenciam como a simulação de um sistema operacional leva a **decisões de projeto não convencionais**, mas que foram essenciais para garantir a funcionalidade e coerência do sistema implementado.

7. Conclusão

O projeto do pseudo sistema operacional alcançou com sucesso os objetivos propostos. Foi possível simular, de maneira didática, os principais componentes de um sistema operacional real: gerenciamento de processos, filas com prioridades, alocação de memória, controle de dispositivos de entrada e saída, além de um sistema de arquivos simplificado.

Como resultado, o sistema se mostrou capaz de gerenciar múltiplos processos com regras de escalonamento além de restrições de recursos. O projeto proporcionou aos integrantes do grupo importantes aprendizados sobre o que tem por trás dos sistemas operacionais.

Referências

[1] Materiais da disciplina e exemplos fornecidos pela Profa. Aletéia Patrícia Favacho de Araújo.