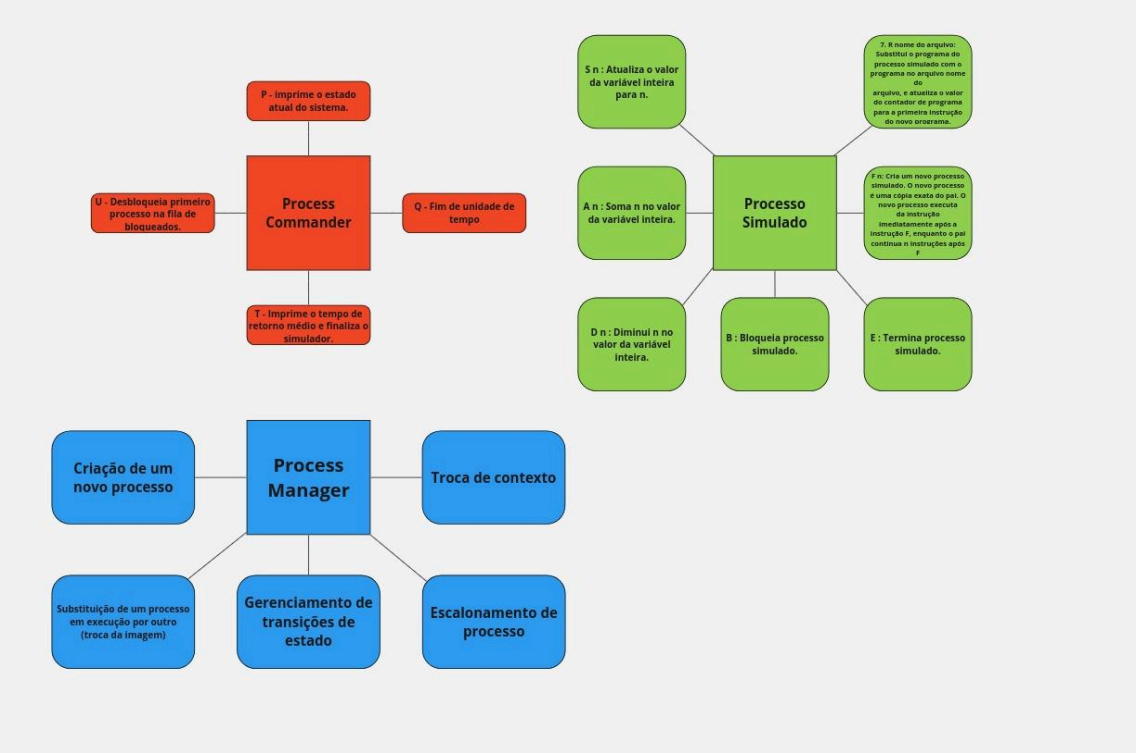
UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
SISTEMAS OPERACIONAIS : 1° semestre de 2024
DOCENTE: Rafael Sachetto Oliveira
DISCENTES : Leonardo Ribeiro; Braian Melo Silva; Igor Augusto de Serpa e Cunha



Sumário:

1. Introdução:

O presente trabalho tem como objetivo o desenvolvimento de um simulador para o gerenciamento de processos, que é capaz de emular cinco funções essenciais associadas à administração de processos em um sistema operacional. As funcionalidades a serem implementadas são as seguintes:

- **Criação de novos processos**: Esta funcionalidade deve permitir a geração de novos processos dentro do simulador, replicando o comportamento de um sistema operacional ao alocar recursos e atribuir identificadores únicos aos novos processos.
- **Substituição do processo atual por um novo processo**: Esta funcionalidade simulará a capacidade do sistema operacional de trocar o processo em execução por outro, garantindo que o estado do processo atual seja preservado e que o novo processo seja corretamente iniciado.

- **Transição de estados de um processo**: Esta funcionalidade permitirá a simulação das diferentes transições de estados que um processo pode experimentar, tais como "pronto", "executando" e "bloqueado". O simulador deverá gerenciar essas transições de forma precisa, buscando refletir os eventos que ocorrem em um sistema operacional real.

- **Escalonamento**: O escalonamento é a função responsável por determinar a ordem de execução dos processos. O simulador deve implementar algoritmos de escalonamento que possam priorizar processos de acordo com algum critério, tal como um sistema operacional faz para otimizar o uso do processador.

- **Troca de contexto**: A troca de contexto é um processo crítico em sistemas operacionais, onde o estado de um processo é salvo e o estado de outro processo é carregado. O simulador deve emular este procedimento, garantindo a integridade e a continuidade da execução dos processos.

Este simulador será desenvolvido utilizando a linguagem de programação C, aproveitando sua eficiência e controle de baixo nível para refletir de maneira fiel as operações realizadas por um sistema operacional.

O principal objetivo deste projeto é proporcionar uma compreensão aprofundada sobre como os sistemas operacionais realizam o gerenciamento de processos. Ao simular estas operações fundamentais, espera-se que os estudantes e profissionais possam obter insights valiosos sobre o funcionamento interno dos sistemas operacionais, aprimorando suas habilidades e conhecimentos na área.

2. Lógica:

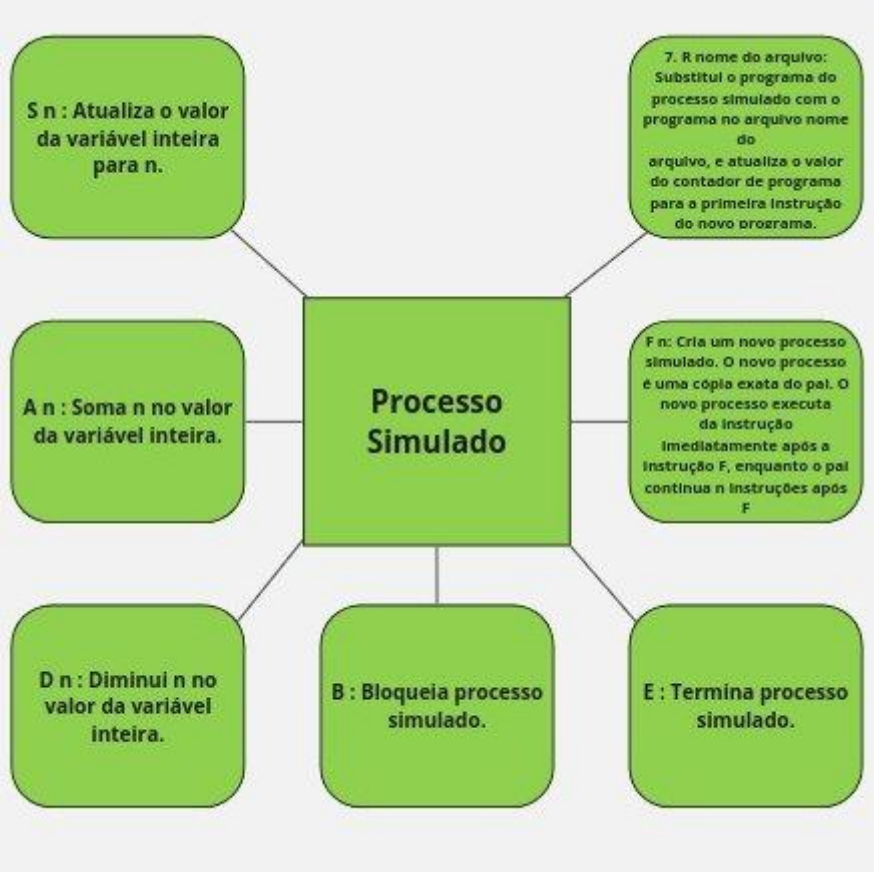
Agora a seguir estão listadas todas as funcionalidades do gerenciador, desde como funcionará os processos até em como os comandos devem ser recebidos pelo terminal.

2. 1 Processo simulado :

Trata-se de um processo que gerencia o valor de uma única variável inteira. Torna-se importante explicar que os processos que serão gerenciados são todos desse tipo.

O primeiro processo é lido de um arquivo denominado “init”, cujo mesmo deve obrigatoriamente estar no mesmo diretório do programa.

Agora abaixo está os tipos de instruções que podem estar nesse arquivo de texto :



A seguir está um exemplo de processo simulado contido no arquivo “init”:

S 217

D 70

A 54

A 10

B

F 1

B

D 35

A 90

R file_e

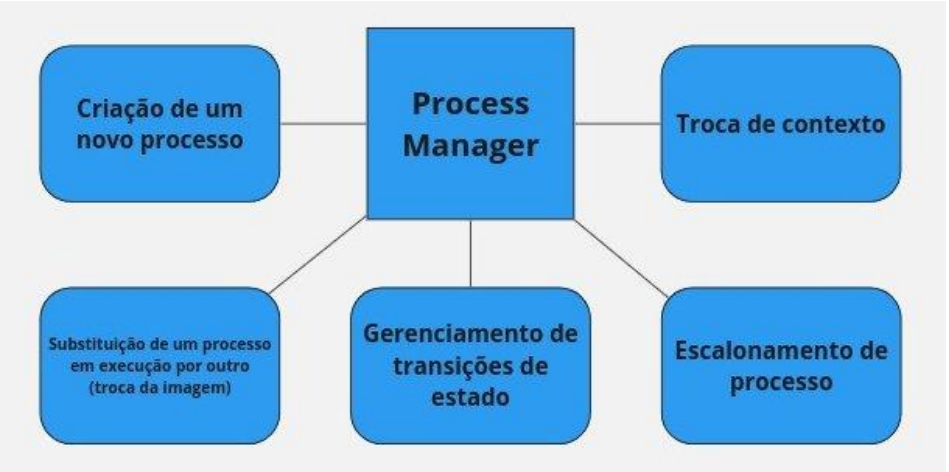
A 7

E

É importante dizer que o único processo criado “do nada” é o processo contido no arquivo “init”, todos os outros processos são criados pela instrução “F” que cria processos filhos.

2.2 Process Manager :

É um conjunto de funcionalidades que realizam o gerenciamento dos processos. É capaz de fazer as seguintes operações :

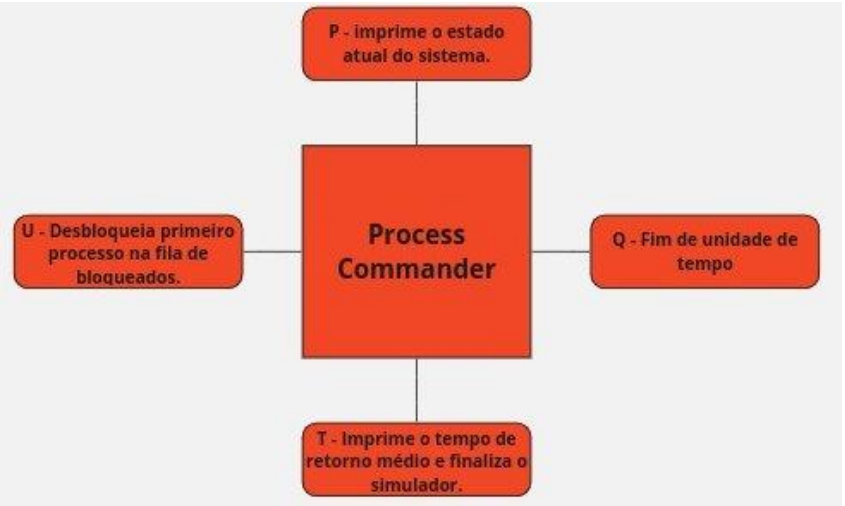


É necessário avisar que as operações que o Process Manager realiza dependem tanto do estado dos processos quanto dos comandos recebidos pelo Process Commander.

2.3 Process Commander :

O Process Commander é responsável pelo envio de comandos para o Process Manager, comandos esses que devem além de passar através de um pipe devem também ser lidos pela entrada padrão (stdin) de forma que um comando lido por segundo.

Os comandos recebidos por ele são :



Além disso o Process Commander usa chamadas de sistema como fork(), sleep(), e pipe().

2.4 Process Reporter :

Em resumo, o Process Reporter serve para imprimir o estado atual do sistema de gerenciamento. A impressão ocorre quando o comando 'P' é recebido pelo Process Commander pela entrada padrão.

2.5 Critério de escalonamento escolhido :

A estratégia de escalonamento escolhida se baseia em um valor de prioridade, quanto maior for a prioridade de um processo mais vezes ele é escalonado. Além disso, processos filhos tem prioridade maior que de seus respectivos pais, e fizemos de forma que não tenha dois processos com o valor de prioridade iguais.

3 Implementação :

Agora a seguir estão alguns detalhes principais acerca da implementação, as TAD's e as funções usadas para a implementação do simulador de processos. Aqui na documentação trataremos apenas de explicar bem superficialmente os inputs e outputs de cada função, pelo simples motivo do código fonte ao todo ter cerca de 700 linhas o que dificultaria não apenas a explicação das funções como também a compreensão do leitor do código fonte. Visando isso, o código fonte (principalmente a

interface) terá a adição de comentários de bloco para caso tenha interesse em se aprofundar na lógica de implementação escolhida.

3.1 Processo Simulado :

Sua implementação pode ser encontrada na interface “processo_simulado.h” e em “processo_simulado.c”.

Primeiramente é preciso dizer que por uma questão de armazenamento foram definidas duas constantes que representam respectivamente o número máximo de instruções que cada processo pode ter, e o número máximo de processos no gerenciador. É importante dizer que caso se interesse por mudar os valores das mesmas não terá problemas.

```
#define MAX_INSTRUCOES 100

#define MAX_PROCESSOS 50
```

A estrutura que representa uma instrução do processo simulado escolhida foi a seguinte :

```
typedef struct Instrucao{

   char instrucao;

   int valor;

   char nomeArq[50];

} Instrucoes;
```

Onde :

- instrucao -> um dos 7 comandos que o processo simulado pode ter.
- valor -> representa (caso exista) um valor numérico de parâmetro de um comando.
- nomeArq -> usado apenas quando o comando recebido é ‘F’, para armazenar o nome do arquivo de texto que substituirá o conteúdo do processo atual.

A estrutura responsável por representar cada processo simulado que será gerenciado escolhida foi a seguinte :

```
typedef struct Processo{

   unsigned prioridade;

   Instrucoes programa[MAX_INSTRUCOES];

   unsigned idPai;

   unsigned idProcesso;

   int variavel;

   unsigned ultimaInstruExec;

   unsigned posiUltimaInstru;

   unsigned tempoInicio;

   unsigned totalCpuUsada;

} Processos;
```

Onde :

- prioridade -> a prioridade do processo.
- programa -> conjunto de instruções/comandos do processo simulado.
- idPai -> id do processo pai.
- idProcesso -> id do processo.
- ultimaInstruExec -> marca o índice da última instrução executada.
- posiUltimaInstru -> representa a quantidade de instruções/comandos que o processo possuí.
- tempoInicio -> usado para representar o valor de tempo em que o sistema se encontrava quando o processo se iniciou.
- totalCpuUsada -> total de cpu usada na execução do processo até o momento atual.

Agora deixarei algumas funções que são bastante intuitivas no seu uso, como dito anteriormente evitaremos deixar detalhes de implementação para deixar mais dinâmica a explicação.

Sendo elas :

cria_processo();

Cria um novo processo.

```
struct Processo *cria_processo(unsigned idProcesso,
      unsigned idPai);
```

Parâmetros:

- idProcesso -> id do processo que será criado.
- idPai -> id do processo pai.

Retorno :

- Ponteiro para um novo processo (inicialmente vazio).

printa_processo();

Printa o estado de um processo.

```
      void printa_processo(struct Processo P);
```

Parâmetros :

- P -> uma variável do tipo processo.

Retorno :

- Nenhum

destroi_processo();

Desaloca a memória ocupada por um processo.

```
      void destroi_processo(struct Processo *P);
```

Parâmetros :

- P -> uma variável do tipo processo.

Retorno :

- Nenhum

inserir_instrucoes_arq();

Insere as instruções de um arquivo de texto em um processo.

```
      void inserir_instrucoes_arq(struct Processo *P,
char *arqNome);
```

Parâmetros :

- P -> uma variável do tipo processo.
- arqNome -> nome/caminho do arquivo com as instruções.

Retorno :

- Nenhum

insere_instru();

Insere uma instrução em um processo.

```
    void insere_instru(struct Processo *P, struct
Instrucao I);
```

Parâmetros :

- P -> uma variável do tipo processo.
- I -> instrução que será inserida.

Retorno :

- Nenhum

executar_instrucao_processo();

Executa a próxima instrução do processo.

```
    int executar_instrucao_processo(struct Processo
**P, unsigned numInstru);
```

Parâmetros :

- P -> uma variável do tipo processo.
- numInstru -> posição da instrução executada.

Retorno :

- Valor referente a instrução executada.

pesquisa_processo();

Pesquisa um processo com base no id em um vetor de processos.

```
    unsigned pesquisa_processo(Processos *P[],
unsigned numProcessos, unsigned id);
```

Parâmetros :

- P -> um vetor de processos.
- numProcessos -> tamanho do vetor de processos.
- id -> id do processo pesquisado..

Retorno :

- Valor referente a posição do vetor em que o processo se encontra, caso o processo não exista retorna a MAX_PROCESSOS.

copia_processo();

Faz uma cópia de um processo.

```
    void copia_processo(struct Processo **pCopia,
struct Processo *pOriginal);
```

Parâmetros :

- pCopia -> processo destino.
- pOriginal -> processo original..

Retorno :

- Nenhum.

finaliza_processo();

Finaliza um processo em um vetor de processos.

```
    unsigned finaliza_processo(Processos *P[],
unsigned numProcessos, unsigned id);
```

Parâmetros :

- P -> um vetor de processos.
- numProcessos -> tamanho do vetor de processos.
- id -> id do processo a ser finalizado.

Retorno :

- Retorna ao novo tamanho do vetor de processos.

3.2 Process Manager :

Sua implementação pode ser encontrada na interface “process_manager.h” e em “process_manager.c”.

O process manager depende de uma série de TAD´s para garantir um funcionamento mais fluido e principalmente tornar seu código mais legível.

A estrutura responsável por representar a CPU é :

```
typedef struct {
   struct Processo *p;
   unsigned idProcessoAtual;
}CPU;
```

Onde :

- p -> processo atual na CPU.
- idProcessoAtual -> id do processo da CPU.

A estrutura responsável por representar a tabela de processos :

```
typedef struct {
   Processos *P[MAX_PROCESSOS];
   unsigned numProcessos;
}TabelaPcb;
```

Onde :

- P -> vetor de processos que estão no gerenciador..
- numProcessos -> quantidade de processos

A estrutura responsável por representar os estados dos processos:

```
typedef struct Estados{
   unsigned idProcessos[MAX_PROCESSOS];
   unsigned numProcessos;
}ProcessosProntos, ProcessosBloqueados,
ProcessosExecutando;
```

Onde :

- idProcessos -> vetor de id’s dos processos pertencentes a esse estado.
- numProcessos -> tamanho do vetor.

A estrutura para representar o tempo :

```
typedef struct {

   unsigned t;

}Tempo;
```

Onde :

- t -> tempo decorrido.

A estrutura para representar o tempo de retorno médio é :

```
typedef struct {

   Tempo tempoRetornoTotal;

   unsigned numProcessosFinalizados;

}TempoRetornoMedio;
```

Onde :

- tempoRetornoTotal -> representa a soma de todos os tempos que levaram para os processos serem concluídos.
- numProcessosFinalizados -> número total de processos que foram finalizados.

E finalmente, a estrutura responsável por representar o Process Manager :

```
typedef struct {

   Tempo tempo;

   CPU *cpu;

   TabelaPcb *pcb;
```

```
    ProcessosProntos pP;
    ProcessosBloqueados pB;
    ProcessosExecutando pE;
    TempoRetornoMedio tRM;
}ProcessManager;
```

Onde :

- tempo -> tempo decorrido.
- cpu -> CPU do simulador.
- pcb -> tabela que contém todos os processos.
- pP -> processos prontos.
- pB -> processos bloqueados.
- pE -> Processos executando.
- tRM -> tempo de retorno médio.

Agora iremos mostrar as funções relacionadas às TAD's apresentadas :

cria_process_manager();

Inicializa a estrutura do Process Manager.

```
ProcessManager *cria_process_manager();
```

Parâmetros :

- Nenhum

Retorno :

- Uma nova estrutura de Process Manager.

destroi_process_manager();

Desaloca a memória alocada para o Process Manager

```
void destroi_process_manager(ProcessManager *Pm);
```

Parâmetros :

- Pm -> process manager.

Retorno :

- Nenhum.

inicia_processo();

Cria/adiciona um novo processo no Process Manager.

```
int inicia_processo(ProcessManager *Pm, char
*arqNome, unsigned idProcesso, unsigned idPai);
```

Parâmetros :

- Pm -> process manager.
- arqNome -> nome/caminho do arquivo que está as instruções do novo processo.
- idProcesso -> id do novo processo.
- idPai -> id do processo pai.

Retorno :

- 1 caso a criação dê alguma falha e 0 caso contrário..

troca_de_contexto();

Realiza a troca de um processo em execução por outro.

```
unsigned troca_de_contexto(ProcessManager *Pm,
unsigned idProcesso);
```

Parâmetros :

- Pm -> process manager.
- idProcesso -> id do novo processo para executar.

Retorno :

- 1 caso a criação dê alguma falha e 0 caso contrário.

troca_de_imagem();

Realiza uma troca de imagem entre um processo que foi interrompido por outro.

```
unsigned troca_de_imagem(ProcessManager *Pm,
unsigned idProcesso);
```

Parâmetros :

- Pm -> process manager.
- idProcesso -> id do novo processo para executar.

Retorno :

- 1 caso a criação dê alguma falha e 0 caso contrário.

escalonamento();

Realiza o método de escalonamento visto anteriormente.

```
void escalonamento(ProcessManager *Pm);
```

Parâmetros :

- Pm -> process manager.

Retorno :

- Nenhum.

process_manager();

Gerencia o sistema de processos com base no comando recebido pelo Process Commander.

```
void process_manager(ProcessManager *Pm, char comando);
```

Parâmetros :

- Pm -> process manager.
- comando -> comando recebido pelo Process Commander.

Retorno :

- Nenhum.

3.3 Process Commander :

Sua implementação pode ser encontrada na interface “commander.h” e em “commander.c”.

O Process Commander depende de uma única função :

commander();

Envia comandos para o Process Manager através de um pipe.Esta função cria um pipe para comunicação entre processos Ela cria um processo filho para executar o ProcessManager e envia comandos, um por segundo, para ele através do pipe. Os comandos são lidos da entrada padrão (stdin) pelo processo pai e escritos no pipe. O processo filho lê os comandos do pipe e os executa através da função

process_manager(). Após enviar todos os comandos, o processo pai espera a execução dos comandos pelo processo filho e finaliza.

```
int commander(ProcessManager *Pm);
```

Parâmetros :

- Pm -> process manager.

Retorno :

- 1 caso a criação dê alguma falha e 0 caso contrário.

3.4 Process Reporter :

Sua implementação pode ser encontrada na interface “reporter.h” e em “reporter.c”.

O Process Reporter se baseia em apenas uma função :

process_reporter();

Exibe o estado do sistema.

```
void process_reporter(ProcessManager *Pm);
```

Parâmetros :

- Pm -> process manager.

Retorno :

- Nenhum.

3.5 Detalhes extras :

Além das funções acima, existem diversas funções mais simples para o auxílio no funcionamento do simulador, porém como são funções que são somente auxílios para funções mais importantes, um grande exemplo disso é a interface “vetor.h”, que trazem operações básicas com vetores, como a remoção e a adição de um valor no vetor. Novamente é importante avisar que mais detalhes de implementação de todas as funções se encontram nas interfaces.

4 Observações :

Agora a seguir traremos algumas pequenas observações acerca do funcionamento do programa, as mesmas são bastante simples :

- Os únicos comandos recebidos diretamente pelo terminal são “Q”, “U”, “P”, “T”; cujo as mesmas funções estão no diagrama do Process Commander.
- O tempo decorrido só é incrementado quando recebe o comando “Q” se existe um processo na CPU ou se existe algum processo na fila de prontos.
- Quando recebe o comando “U“ o processo não é escalonado imediatamente, ele só é movido para a fila de prontos.
- Quando o comando recebido é “T” ele imprime o tempo de retorno médio e finaliza o programa. Para isso ao menos um processo deve ser finalizado, caso contrário imprime uma mensagem de erro.
- Acompanhado do código fonte deixaremos um exemplo de processo no arquivo denominado “init”, vale lembrar novamente que o arquivo “init” deve estar no mesmo diretório do executável.
- Deixaremos também um executável denominado “gerador”, o mesmo é capaz de gerar exemplos de processos para a execução, caso se interesse basta executar o mesmo.
- O Process Reporter além de printar o estado atual do sistema como um todo, também exibirá todos os detalhes do processo que está na CPU.

5 Conclusão :

Este trabalho apresentou o desenvolvimento de um simulador de gerenciamento de processos, com o objetivo de emular as principais funcionalidades de um sistema operacional.

Desenvolvido em linguagem C, o simulador busca oferecer uma compreensão detalhada do gerenciamento de processos em sistemas operacionais. O foco foi proporcionar um ambiente de aprendizado onde estudantes e profissionais pudessem explorar e entender as operações fundamentais de um sistema operacional real. Através das funcionalidades descritas, o simulador oferece uma plataforma prática para o estudo e análise do comportamento dos processos e da administração de recursos.

Em suma, este simulador não só proporciona um ambiente prático para a aprendizagem e exploração dos conceitos de gerenciamento de processos, mas também serve como uma ferramenta valiosa para aprofundar o entendimento sobre o funcionamento interno dos sistemas operacionais. Ao replicar operações

fundamentais, espera-se que os usuários adquiram conhecimentos valiosos e aprimorem suas habilidades na área de sistemas operacionais.