Relatório Técnico — Mini-Shell (Prática SO)

Dupla:

- Andrey Henrique de Abreu Carneiro - 555740
- Cauã Nobre Fagundes Moura - 553387

Dificuldades Encontradas e Soluções Aplicadas

1. **Gerenciamento de processos em background**
    - **Dificuldade:** O controle de múltiplos processos rodando simultaneamente. Era necessário armazenar e monitorar os PIDs dos processos filhos executados em segundo plano, exibindo o número do *job* e removendo-os quando terminassem.
    - **Solução:** Foram criados os arrays `bg_pids[]` e `bg_jobs[]` para armazenar o PID e o ID do job, respectivamente. Funções auxiliares como, `remove_bg_process()` foram implementadas para gerenciar a lista de processos e limpar os que já haviam terminado com o uso de `waitpid()` e a flag `WNOHANG`.
2. **Implementação de comandos internos**
    - **Dificuldade:** Foi necessário diferenciar comandos internos (`exit`, `pid`, `jobs`, `wait`) de comandos externos.
    - **Solução:** A função `is_internal_command()` verifica se o comando digitado corresponde a algum desses, e `handle_internal_command()` executa a lógica apropriada para cada caso.
3. **Sincronização no comando `wait`**
    - **Dificuldade:** O comando `wait` precisava bloquear o shell até que todos os processos em background terminassem.
    - **Solução:** A função `wait_all_bg()` foi criada para percorrer todos os processos ativos e aguardar seu término utilizando `wait()`, garantindo que o shell só continuasse após todos finalizarem.

Testes Realizados e Resultados Obtidos

Testes Básicos

Comando	Resultado Esperado	Resultado Obtido
`ls`	Lista o conteúdo do diretório atual	Funcionou corretamente

`date`	Mostra a data e hora do sistema	Funcionou corretamente
`pid`	Mostra o PID do shell e do último processo filho	Exibiu corretamente os dois valores
`exit`	Encerra o shell	Encerrou o shell após aguardar processos em background

Testes Avançados

Comando	Descrição	Resultado Obtido
`sleep 10 &`	Executa em background e mostra `[1] 1234`	Funcionou corretamente
`jobs`	Lista processos ativos em background	Listou todos os processos com status “Running”
`sleep 5 &`	Adiciona outro processo em background	Novo job exibido com ID e PID distintos
`wait`	Aguarda todos os processos em background	Exibiu mensagens `[n]+ Done` e finalizou todos corretamente
`jobs` após `wait`	Lista vazia de processos	“Nenhum processo em background” foi exibido corretamente

Observação: Erros como comandos inexistentes foram tratados com mensagens adequadas via `perror("Erro no execvp")`.

Análise dos Conceitos de SO Aplicados

Abaixo estão os principais conceitos aplicados:

1. Criação de Processos — fork()
   O uso de fork() permitiu criar um novo processo (filho) a partir do shell (pai). Cada comando externo é executado por um processo filho independente. Essa chamada duplica o processo pai, criando dois fluxos de execução distintos.
2. Substituição da Imagem de Processo — execvp()
   Após a criação do filho, ele substitui sua imagem pelo programa solicitado através de execvp(). Isso demonstra o conceito de substituição da imagem do processo, essencial para a execução de novos programas no Unix.
3. Sincronização de Processos — wait() e waitpid()
   A sincronização entre o processo pai (shell) e seus filhos é feita com wait() (foreground) e waitpid(..., WNOHANG) (background). Esses mecanismos garantem que o shell controle o fluxo de execução e evite processos zumbis.
4. Identificação de Processos (PIDs)
   O uso de getpid() e waitpid() reforça o conceito de identificador de processo (PID), essencial para distinguir e monitorar cada execução individual.
5. Processos em Foreground e Background
   A implementação do símbolo & exemplifica o gerenciamento de múltiplos processos concorrentes. No foreground, o shell aguarda o término do processo. No background, ele apenas registra o PID e permite ao usuário continuar usando o shell.
6. Evitação de Processos Zumbis
   Ao usar waitpid(-1, &status, WNOHANG) dentro de clean_finished_processes(), o shell garante a coleta do status de terminação dos filhos, impedindo a existência de processos zumbis no sistema.
7. Sinais e Tratamento de Interrupções
   O uso de signal(SIGINT, SIG_DFL) no processo filho restaura o comportamento padrão do sinal de interrupção, evitando que o filho herde o tratamento do pai e garantindo comportamento independente.