Prática Mini-Shell

Sistemas Operacionais

Integrantes:
Antônio Gabriel - 539628
Erik Bayerlein - 537606
Júlia Naomi - 538606

Sumário

Dificuldades encontradas e soluções aplicadas

Linguagem C

A principal dificuldade foi o domínio da linguagem C. Isso ficou bastante evidente quando tentamos lidar com array de strings ou implementar certos conceitos.

Dessa forma, foi necessário recorrer a pesquisas e práticas para compreender melhor os problemas. Além disso, tanto o livro do professor Carlos Maziero quanto o documento de especificação da prática foram fundamentais para orientar nos orientar.

Ademais, muitas vezes sabíamos o que precisava ser feito, mas, uma vez que não tínhamos domínio da linguagem escolhida, não conseguimos implementar de forma satisfatória.

Por fim, a implementação de testes automatizados para as funções utilizadas na main() também foi um desafio. Contudo, após algumas pesquisas, optamos por utilizar a biblioteca Check.

`

Manipulação de strings

Lidar com string foi, de fato, um problema que enfrentamos inicialmente, principalmente devido a comparações de tipos e ponteiros para memória.

Exemplo de problemas enfrentados:
- realizar um split da string (char*) e salvar no array
- comparar strings (variável do tipo char* com "string")

```
args[i] = strtok(input, delim);
while (args[i] != NULL)
  args[++i] = strtok(NULL, delim);
```

Memória entre processos

No trabalho de lidar com as variáveis globais last_child_pid e bg_processes, inicialmente os valores não estavam sendo armazenados corretamente. Por esse motivo, ao executar os comandos jobs ou pid, a saída não correspondia ao esperado.

A solução aplicada foi baseada no documento de especificação. A partir dele, conseguimos compreender o funcionamento esperado e ajustar a implementação para que os valores das variáveis fossem armazenados corretamente.

A princípio, optamos por utilizar a chamada de sistema execve(). No entanto, não sabíamos que ela substituiria o processo atual pelo novo programa. Ou seja, o processo filho deixa de ser o mesmo após a execução do execve() e passa a ser inteiramente o programa chamado. Como consequência, o processo filho perde todas as variáveis e estados que possuía antes da chamada.

Isso gerou um grande problema, já que precisávamos manter a lista de PIDs atualizada. Como o processo filho perde todas as variáveis e estados após a chamada do execve(), tornou-se impossível preservar essa lista dentro dele.

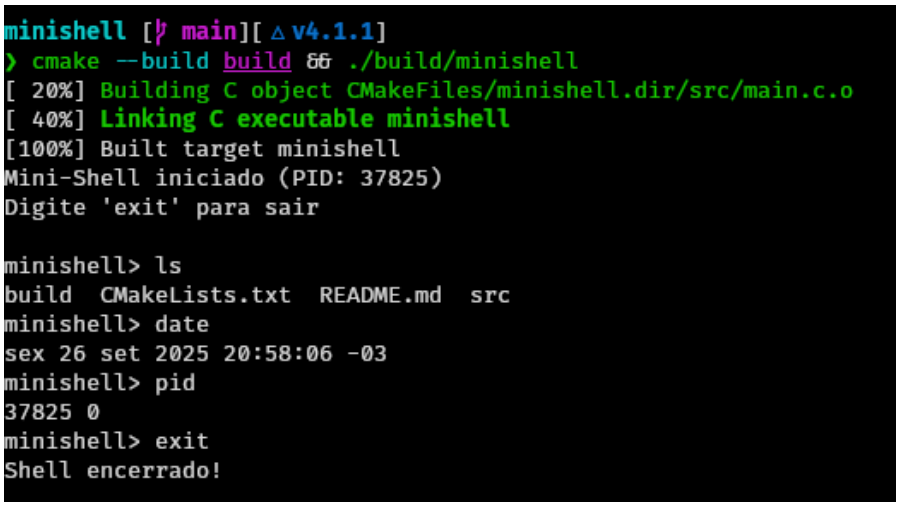
Testes realizados e resultados obtidos

Utilizamos os inputs presentes no documento de especificação da prática, além de testes adicionais criados pela equipe através da biblioteca de teste Check.

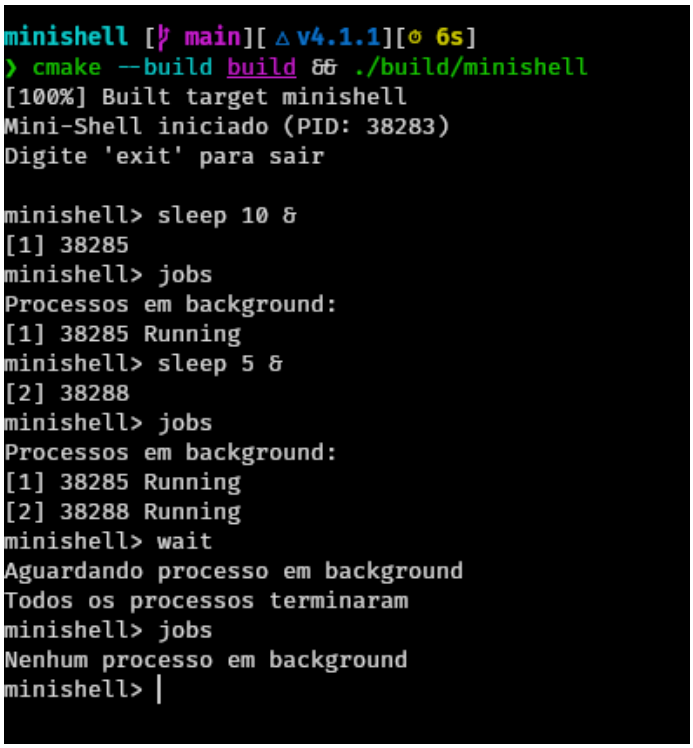
Testes manuais

Nos testes básicos foram avaliados comandos internos e externos sem lidar com subprocessos. Já nos testes avançados foram testados os comandos internos e o comportamento ao lidar com subprocessos e funções padrões do POSIX como wait e jobs.

- Testes básicos:

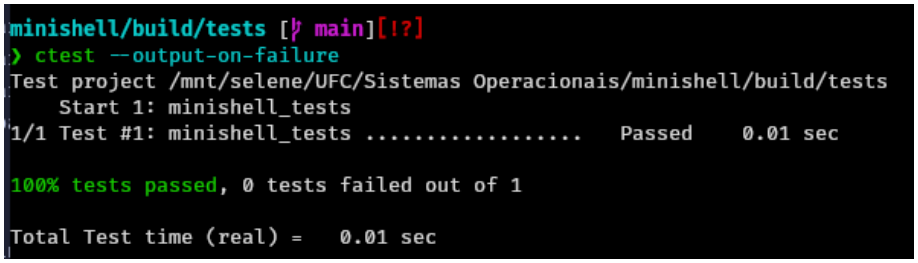


- Testes avançados:



Testes automatizados

Além dos testes manuais, implementamos testes automatizados utilizando a biblioteca Check, que permitiram validar o comportamento esperado do shell.




```c
void setup(void) {
  bg_count = 0;
  memset(bg_processes, 0, sizeof(bg_processes));
}

START_TEST(test_parse_command_basic) {
  char input[] = "ls -l";
  char *args[MAX_ARGS];
  int background = 0;
  parse_command(input, args, &background);

  ck_assert_str_eq(args[0], "ls");
  ck_assert_str_eq(args[1], "-l");
  ck_assert_ptr_eq(args[2], NULL);
  ck_assert_int_eq(background, 0);
}
END_TEST
{...}
int main(void) {
  int number_failed;
  Suite *s;
  SRunner *sr;

  s = minishell_suite();
  sr = srunner_create(s);

  srunner_run_all(sr, CK_VERBOSE);
  number_failed = srunner_ntests_failed(sr);
  srunner_free(sr);
  return (number_failed == 0) ? 0 : 1;
}
```

Análise dos conceitos de SO aplicados

Conceitos aplicados:

- Processos em background e foreground: as chamadas de sistema fork() e exec() foram utilizadas, respectivamente, com o objetivo de criar um novo processo e substituir o espaço de memória do processo filho com um novo programa. O wait() foi empregado para que o processo pai aguardasse a conclusão do processo filho antes de continuar sua execução, garantindo a sincronização.

- Processos filhos: A utilização de uma lista de PID para gerenciar os processos filhos em background foi importante para o controle e monitoramento dos processos em execução, evitando a criação de processos zumbis.

- Parsing e interpretação de comandos: A utilização da API strtok e execvp para dividir a string de entrada e interpretar o comando foi essencial para a funcionalidade do minishell.

Questões de reflexão

Como os Process IDs (PIDs) diferem entre processos pai e filho?

Após executar comandos em background, foi observado que o PID do filho é diferente do PID do pai, e geralmente maior. Isso permite deduzir a ordem de criação dos processos apenas comparando seus PIDs.

Por que as variáveis têm valores diferentes nos dois processos em fork-print.c?

Como se tratam de processos diferentes, o sistema operacional aloca áreas de memória separadas para cada um. Assim, após o fork(), qualquer alteração feita em variáveis existe apenas no processo que a realizou.

O que acontece com o processo filho após a chamada execve()?

O processo filho entra em estado de execução do novo programa, deixando de ser o mesmo que era antes do execve(). Ele perde todas as variáveis e estados anteriores. Já com execvp(), o processo filho continua sendo o mesmo, apenas executando um novo programa enquanto mantém suas variáveis e estados.