Atividade Prática: Mini-Shell

Nome	Antonio Gabriel	Erik Bayerlein	Julia Naomi
Matrícula	539628	<define>	538606

Sumário

1) Dificuldades encontradas e soluções aplicadas 2
   1.a) Linguagem C 2
   1.b) Manipulação de strings 2
   1.c) Memoria entre processos. 2
2) Testes realizados e resultados obtidos 2
   2.a) Teste manuais: 3
      2.a.i) Testes basicos: 3
      2.a.ii) Testes avançados 3
      2.a.iii) Testes automatizados: 4
3) Análise dos conceitos de SO aplicados 5
   3.a) Sobre as questoes de reflexão 5
      3.a.i) Como os Process IDs (PIDs) diferem entre processos pai e filho? 5
      3.a.ii) Por que as variáveis têm valores diferentes nos dois processos em fork-print.c? 5
      3.a.iii) O que acontece com o processo filho após a chamada execve()? 5

1) Dificuldades encontradas e soluções aplicadas

1.a) Linguagem C

Os membros da equipe tiveram uma dificuldade com `C` como por exemplo lidar com array de strings ou como implementar certos conceitos.

A solução aplicada foi pesquisa e muita prática, evitamos o uso de LLM's para resolver problemas de código, nos forçando a pesquisar e entender o problema, por exemplo, sabíamos o que precisávamos fazer mas não como fazer em C, por isso pesquisamos como fazer e testamos implementação, além disso o documento de especificação da pratica auxiliou muito.

Semelhante a isso, implementar testes automatizados paras funções utilizadas na `main()`, também foram um desafio, por recomendação de um amigo, optamos por utilizar a biblioteca check. Apos algumas pesquisas de como configurar e implementar testes automatizados em C, os testes foram implementados com sucesso, como se pode ver na Seção 2.a.iii

1.b) Manipulação de strings

Lidar com string foi o pequeno problema que enfrentamos inicialmente devido a comparações de tipos e ponteiros para memória, mas como dito no Seção 1.a, após algumas pesquisas em sites como stackoverflow

Um dos exemplos enfrentados foram:

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

Novamente, como dito na Seção 1.a, a solução aplicada foi soluções encontradas no stackoverflow com sugestões de como implementar, por exemplo, "how to split string in C", "how to split string and save in an array".

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

1.c) Memoria entre processos.

Na parte do trabalho de lidar com as variáveis globais, last_child_pid e bg_processes, a principio o valor não estava sendo salvo corretamente, por isso quando executava o comando jobs ou pid, o output não estava como deveria.

A solução aplicada veio do documento de especificação, através dele conseguimos entender

A principio escolhemos utilizar a metodologia do execve mas o que não sabíamos é que o execve substitui o processo atual pelo novo processo, ou seja, o processo filho deixa de ser o mesmo que era antes do execve() e passa a ser o programa que foi chamado pelo execve(), por conta disso, o processo filho perde todas as variáveis e estados que tinha antes do execve(). Isso nos trouxe um grande problema, pois precisamos manter a lista de pids atualizada, e como o processo filho perde todas as variáveis e estados que tinha antes do execve(), não conseguimos manter a lista de pids atualizada.

2) Testes realizados e resultados obtidos

Para testar o minishell foi utilizado os input presentes no documento de especificação da prática, além de testes adicionais criados pela equipe. utilizando a biblioteca de teste Check.

2.a) Teste manuais:

Na Seção 2.a.i testes para função de comandos internos e externos mas sem lidar com subprocessos. Já na Seção 2.a.ii foi testado os comandos internos e o comportamento ao lidar com subprocessos e funções padrões do POSIX como wait e jobs.

2.a.i) Testes basicos:

```
minishell [ main][ △ v4.1.1]
❯ cmake --build build && ./build/minishell
[ 20%] Building C object CMakeFiles/minishell.dir/src/main.c.o
[ 40%] Linking C executable minishell
[100%] Built target minishell
Mini-Shell iniciado (PID: 37825)
Digite 'exit' para sair

minishell> ls
build  CMakeLists.txt  README.md  src
minishell> date
sex 26 set 2025 20:58:06 -03
minishell> pid
37825 0
minishell> exit
Shell encerrado!
```

Figura 1: execução dos testes básicos

2.a.ii) Testes avançados

```
minishell [ main][ △ v4.1.1][ 6s]
❯ cmake --build build && ./build/minishell
[100%] Built target minishell
Mini-Shell iniciado (PID: 38283)
Digite 'exit' para sair

minishell> sleep 10 &
[1] 38285
minishell> jobs
Processos em background:
[1] 38285 Running
minishell> sleep 5 &
[2] 38288
minishell> jobs
Processos em background:
[1] 38285 Running
[2] 38288 Running
minishell> wait
Aguardando processo em background
Todos os processos terminaram
minishell> jobs
Nenhum processo em background
minishell>
```

Figura 2: execução dos testes avançados

2.a.iii) Testes automatizados:

```
minishell/build/tests [ main][!?]
❯ ctest --output-on-failure
Test project /mnt/selene/UFC/Sistemas Operacionais/minishell/build/tests
    Start 1: minishell_tests
1/1 Test #1: minishell_tests ..................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.01 sec
```

Figura 3: execução dos testes automatizados

```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;
}
```

Listagem 1: trecho do código de testes

3) Análise dos conceitos de SO aplicados

Conceitos aplicados:

- processos em background e foreground:

as chamadas de sistema `fork()` e `exec()` que foram utilizadas, respectivamente, para criar um novo processo e para substituir o espaço de memória do processo filho com um novo programa. o `wait()` foi utilizado para que o processo pai aguardasse a conclusão do processo filho antes de continuar sua execução, garantindo a sincronização entre os processos.

- Processos filhos:

A utilização de uma lista de PID para gerencias os processos filhos em background foi importante para o controle e monitoramento dos processos em execução, evitando a criação de processos zumbis, garantindo que todos os processos filhos sejam devidamente aguardados e finalizados.

- Parsing e interpretação de comandos:

a uso da API strtok e execvp para dividir a string de entrada e interpretar o comando foi essencial para a funcionalidade do minishell, permitindo que o shell entenda e execute os comandos fornecidos pelo usuário de forma eficiente.

3.a) Sobre as questoes de reflexão

3.a.i) Como os Process IDs (PIDs) diferem entre processos pai e filho?

Ao analisar o comportamento apos executar comandos em background no minishell, foi observado que o PID do filho é diferente do PID do pai e além disso, o PID do filho é maior que o do pai, o que é possível deduzir que um processo veio depois do outro apenas comparando os PIDs

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

Como são processos diferentes, são como instancias diferente do mesmo programa, por conta disso, o S.O. aloca espaços diferentes de memoria para cada um, ao fazer isso, os valores das variáveis, caso sejam alteradas apos o `fork()`, terão valores diferente em cada processo

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

O processo filho entra em estado de execução do novo programa, ou seja, o processo filho deixa de ser o mesmo que era antes do `execve()` e passa a ser o programa que foi chamado pelo `execve()`, por conta disso, o processo filho perde todas as variáveis e estados que tinha antes do `execve()`.

Enquanto se executado com `execvp()`, o processo filho continua sendo o mesmo, apenas executando um novo programa, mas mantendo suas variáveis e estados.