Lab 4

Neste relatório seguimos o caminho da nossa progressão, portanto ele não é paralelo ao fluxo do código.

Primeiramente pensamos que era necessário um loop infinito para ler, parsear e executar os comandos, e também uma função para “limpar” o terminal e imprimir o prompt da nossa shell, assim começamos nossa função main.

Clear()

Foi a primeira função a ser escrita. Ela limpa a tela do terminal com um printf.

PrintaCaminho()

Inicialmente usamos a função getcwd() para pegarmos o diretório atual e usarmos como prompt, porém percebemos que no terminal só é incluído o último diretório do caminho, assim fizemos a função UltimoElemCaminho como solução. Também decidimos colorir a impressão para facilitar a experiência do usuário ao digitar os comandos.

UltimoElemCaminho()

Fizemos essa função para ter somente o diretório atual no caminho da Shell. Desse modo, nossa tela não ficava tão poluída de informação e conseguiamos saber onde estávamos.

LeInput()

Decidimos usar ponteiro de char para ler a entrada já que C não possui string. Passamos esse ponteiro para a função LeInput(), para que a função saiba aonde armazenar o input do usuário.

Inicialmente usamos um scanf para ler a entrada, porém após pesquisas descobrimos que não era o ideal para programas complexos pois é muito sujeito a falha. Assim, optamos por um loop de getchar() que se encerra quando encontra um EOF ou ‘\n’.

No início do loop temos uma constante BUFFERLIMITE que é o tamanho que usamos para alocar a nossa string. Quando a entrada atinge esse limite, fazemos um realloc acompanhando o excedente que tivemos do BUFFERLIMITE a cada realloc.

ParsedInput()

Como um comando pode ser acompanhado de argumentos, esses separados por um espaço, precisamos quebrar nossa string de entrada para ter acesso a cada parte individualmente.

Na função ParseInput() recebemos a string de entrada do usuário, onde armazenaremos nossa string quebrada (em um double pointer de char) e o nosso separador, no caso ‘ ‘.

Usamos a função strtok() passando o separador e armazenando os tokens (partes do comando) na nossa array de string, ao final do parseamento adicionamos um NULL para determinar o fim do array.

Inicialmente, tentamos reconhecer se o comando digitado era algum comando mencionado do escopo do lab, através de vários if's, porém tivemos uma problemática, pois quando o comando não era built-in a shell era encerrada.

Para solucionar esse problema pensamos em executar os comandos através de um filho, isso resultou em outro problema, dessa vez o contrário, não conseguiamos sair da shell.

Concluímos então que comandos não built-in deveriam ser executados pelo filho, e comando built-in deveriam ser executados pelo pai.

EhBuiltin()

Para facilitar a checagem dos comandos, fizemos a função EhBuiltin() que retornava 1 caso o comando digitado do usuário fosse built in e 0 caso não fosse, assim podiamos trata-los com mais facilidade.

ExecNaoBuiltin()

Pesquisamos como executar comandos dentro de um programa e descobrimos a família de funções exec.

Usamos o execvp() passando o comando digitado pelo usuário e seus argumentos. A função exec termina o processo após sua execução, por isso executamos ele pelo filho.

Caso ocorra um erro, o filho encerra e é impresso uma mensagem de erro.

ExecBuiltin()

Usamos uma sequência de if's para encontrar o comando e chamar as funções apropriadas. Essa função é executada pelo pai.

Cd()

Como o comando cd necessariamente é acompanhado de argumentos, recebemos esses parâmetros para a função. Se o caminho não foi passado, o que recebemos será um NULL, assim tratamos através de um aviso ao usuário e retorno a shell.

Caso passe dessa validação, usamos a função chdir(), para trocar para o diretório que o usuário indicou, tratando também com um aviso ao usuário e retorno a shell caso o chdir falhe.

O próximo passo que tomamos foi implementar os jobs.

Para isso, inicialmente pensamos no formato da struct para armazenar os dados do processo. Determinamos os campos nome (índice do job), id (id do job), modo (background ou foreground), status (Executando, Parado, Concluído ou Terminado), pid (id do processo), flag (variável auxiliar para sabermos se foi terminado recentemente).

Para armazenarmos todos os jobs e podermos acessá-los em qualquer função fizemos uma array global (jobsCriados). Começamos a implementar o jobs através de 3 funções essenciais: AdicionaJobs, EliminaJobs e ListJobs.

AdicionaJobs()

Nessa função recebemos o modo que o usuário determinou e o nome do comando. Com esses dados e o pid do processo atual (g_pid_crp) criamos um novo job e armazenamos na nossa lista global.

EliminaJobs()

Essa função recebe o pid do processo que foi interrompido, percorre o array com todos os jobs até achar o que foi recebido. O status do job é alterado para “Terminado”, caso seja um processo em background imprime seus dados, e por fim, mata o processo através de um sinal SIGKILL.

ListJobs()

Essa função percorre o array com todos os jobs imprimindo no formato que vimos que o terminal imprimia.

SigHandler()

Após implementar a estrutura do jobs começamos a tratar sinais enviados pelo usuário para terminar ou suspender os processos.

Nesse momento lembramos das funções vistas nas listas, signal() e kill(), e usamos elas para tratar e enviar sinais no SigHandler.

Na recepção desses sinais esperávamos algum filho que houvesse sido suspenso ou terminado. O SIGCHLD era ativado e tinhamos acesso ao pid do filho que entrou nessa condição. Dentro do SIGCHLD conseguimos verificar o porquê do filho ter entrado no if e assim tratar o caso e retornar.

Acessamos o pid do filho através do SIGCHLD e verificamos o motivo de ter ativado o sinal com as funções WIFEXITED, WIFSIGNALED, WIFSTOPPED e WIFCONTINUED.

WIFEXITED é ativada se o filho terminou o processo sem nenhuma interrupção, chamando a função ConcluiJobs().

WIFSIGNALED é ativada se o filho tiver sido terminado por um sinal, apenas retornando já que esse sinal já foi tratado pelo EliminaJobs().

WIFSTOPPED é ativada se o filho é suspenso, apenas retornando já que esse sinal já foi tratado pelo SuspendeJobs()

WIFCONTINUED é ativada caso o filho receba um SIGCONT, nesse caso só retornamos pois tratamos esse caso mais a frente.

ConcluiJobs()

Fizemos essa função pois percebemos que havia o caso do processo terminar normalmente, sendo assim não era necessário mata-lo como ocorre no EliminaJobs(), apenas mudar seu status.

SuspendeJobs()

Fizemos essa função pois era necessário tratar sinais de parada, mudando o status e enviando um sinal ao processo.

Bg()

Para colocar o processo em background criamos mais duas funções, pois reparamos que bg poderia ser usado com ou sem argumento (o índice do processo). Sendo assim, tivemos a origem de: CrProcessToBg() e ProcessToBg()

CrProcessToBg()

É uma função que recebe um argumento nulo, colocando o último processo como background. Percebemos que o processo atual só pode ir para o background quando ele está parado, pois só assim o usuário tem o controle do terminal. Portanto, precisamos enviar um sinal para o processo continuar.

ProcessToBg()

É uma função que recebe um índice, colocando o processo de id equivalente para Executar no background. Esse só ocorrerá caso o status do processo escolhido seja “Parado”, pois somente assim o usuário teria acesso a shell e nos certificaríamos que não estamos tentando colocar um processo já Concluido ou Terminado para rodar.

Fg()

Para tratarmos o comando fg escrevemos a função Fg(), que checava se o usuário havia digitado algum argumento, já que para essa função é necessário, e chamava a função ProcessToFg que efetivamente executaria o comando.

ProcessToFg()

Dentro dessa função nos certificamos que o processo de índice escolhido poderia ser colocado em Fg. Para isso, comparamos o status do processo escolhido com “Executando” ou “Parado”, se for correspondente alteramos o status para “Executando” e setamos o pgid desse processo para foreground. No fim, mandamos um sinal para o filho voltar a ser executado. E aqui fica um dos únicos problemas que não conseguimos resolver, pois não conseguimos devolver o controle do terminal para o processo, fazendo ele executar em background.

Seguimos então para os comandos built-in que faltavam, kill e help.

Kill()

Para implementar o kill foi relativamente fácil como já tinhamos quase o todo código rodando. Assim, apenas percorremos o vetor de jobs até achar o processo que usuário quer matar, verificando se ele não já foi terminado ou concluído e mandando um sinal SIGKILL para o processo.

Help()

No help apenas damos informações dos processos built-in que implementamos e como acessar mais detalhes sobre os não built in