OS lab Assignment – 2

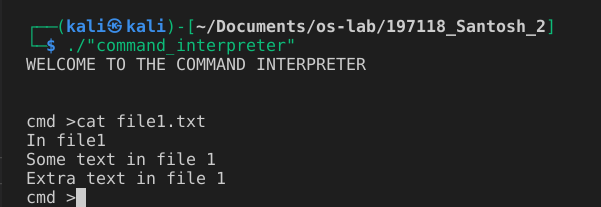
B Santosh Kumar 197118 CSE-A

Implement command interpreter, read a line of command, integrate IO redirection and

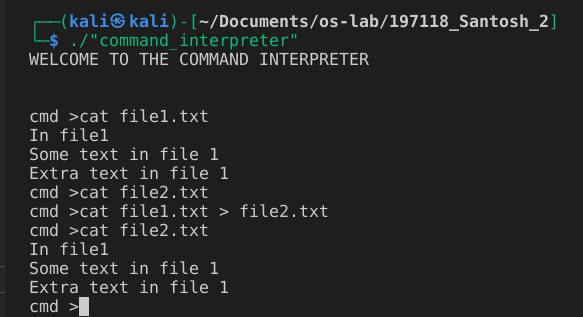
Piping.

|  |
| --- |
| #include <errno.h>  #include <fcntl.h>  #include <linux/limits.h>  #include <stdbool.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/stat.h>  #include <unistd.h>  #define max(a, b) \      ({ \_\_typeof\_\_ (a) \_a = (a); \         \_\_typeof\_\_ (b) \_b = (b); \       \_a > \_b ? \_a : \_b; })  void printWelcome() {      printf("WELCOME TO THE COMMAND INTERPRETER\n\n\n");  }  void splitInputIntoWords(char \*input, int len, char \*\*\*allWords, int \*argumentsLength, char \*delimiter) {      char \*word;      char \*savePointer1 = NULL;      char \*savedInput1 = (char \*)calloc(len + 1, sizeof(char));      char \*savedInput2 = (char \*)calloc(len + 1, sizeof(char));      strcpy(savedInput1, input);      strcpy(savedInput2, savedInput1);      savedInput1[len] = savedInput2[len] = '\0';      char \*deLimit = delimiter;      word = strtok\_r(savedInput1, deLimit, &savePointer1);      int countNoOfWords = 0;      int maxLenWord = 0;      while (word != NULL) {          int len = strlen(word);          maxLenWord = max(len, maxLenWord);          word = strtok\_r(NULL, deLimit, &savePointer1);          countNoOfWords++;      }      for (int i = 0; i < (\*argumentsLength); i++) {          free((\*allWords)[i]);      }      if ((\*allWords)) free(\*allWords);      (\*allWords) = NULL;      (\*argumentsLength) = 0;      (\*allWords) = (char \*\*)calloc(countNoOfWords + 1, sizeof(char \*));      for (int i = 0; i < countNoOfWords; i++) {          (\*allWords)[i] = (char \*)calloc(maxLenWord + 1, sizeof(char));      }      (\*allWords)[countNoOfWords] = NULL;      char \*savePointer2 = NULL;      word = strtok\_r(savedInput2, deLimit, &savePointer2);      int index = 0;      while (word != NULL) {          int currLen = strlen(word);          for (int i = 0; i < currLen; i++) {              (\*allWords)[index][i] = word[i];          }          (\*allWords)[index][currLen] = '\0';          word = strtok\_r(NULL, deLimit, &savePointer2);          index++;      }      (\*argumentsLength) = countNoOfWords;      return;  }  bool checkIfFile(char \*path) {      struct stat buffer;      int exists = stat(path, &buffer);      if (exists == 0 && (S\_IFREG & buffer.st\_mode)) {          return 1;      }      return 0;  }  void checkInCurrDir(char \*commandName, char \*\*filePath) {      if (checkIfFile(commandName)) {          char \*temp = NULL;          filePath = realpath(commandName, (temp));      }      return;  }  void checkInPATH(char \*commandName, char \*\*filePath) {      char \*tempPath = getenv("PATH");      char \*fullPath = (char \*)calloc(strlen(tempPath), sizeof(char));      strcpy(fullPath, tempPath);      char \*token = strtok(fullPath, ":");      struct stat buffer;      while (token != NULL) {          char \*currItem = (char \*)calloc(strlen(commandName) + strlen(token), sizeof(char));          sprintf(currItem, "%s/%s", token, commandName);          if (checkIfFile(currItem)) {              char \*temp = NULL;              (\*filePath) = realpath(currItem, (temp));              return;          }          token = strtok(NULL, ":");      }      return;  }  void getActualPath(char \*commandName, char \*\*filePath) {      checkInCurrDir(commandName, filePath);      if ((\*filePath) != NULL) return;      checkInPATH(commandName, filePath);      return;  }  void removeWhiteSpaces(char \*str) {      int i = 0, j = 0;      while (str[i] != '\0') {          if (str[i] != ' ' && str[i] != '\n')              str[j++] = str[i];          i++;      }      str[j] = '\0';      return;  }  void segregateFiles(char \*command, int noOfFiles[], char \*\*allFiles[]) {      char \*savedInput1 = (char \*)calloc(strlen(command), sizeof(char));      char \*savedInput2 = (char \*)calloc(strlen(command), sizeof(char));      strcpy(savedInput1, command);      strcpy(savedInput2, command);      int index = 0;      int maxLen = 0;      int lenOfString = strlen(command);      for (; index < lenOfString; index++) {          if (savedInput1[index] == ' ' || savedInput1[index] == '\t' || savedInput1[index] == '\n') continue;          int j = index;          int count = 0;          for (j = index; j < lenOfString; j++) {              if (savedInput1[j] != '>') {                  if (savedInput1[j] == '<') j++;                  break;              }              count = count + 1;          }          if (count > 3) {              printf("There is some error here %s\n", command);              exit(0);          }          while (++j && j < lenOfString) {              if (savedInput1[j] == ' ' || savedInput1[j] == '\t' || savedInput1[j] == '\n')                  continue;              else                  break;          }          index = j;          noOfFiles[count]++;          while (j + 1 < lenOfString && savedInput1[j + 1] != ' ' && savedInput1[j + 1] != '>' && savedInput1[j + 1] != '<') j++;          maxLen = max(maxLen, j - index + 1);          index = j;      }      for (int i = 0; i < 3; i++) {          allFiles[i] = (char \*\*)calloc(noOfFiles[i], sizeof(char \*));          for (int j = 0; j < noOfFiles[i]; j++) {              allFiles[i][j] = (char \*)calloc(maxLen + 1, sizeof(char));          }      }      index = 0;      int temp[3] = {0};      for (; index < lenOfString; index++) {          if (savedInput1[index] == ' ' || savedInput1[index] == '\t' || savedInput1[index] == '\n') continue;          int count = 0;          int j = index;          for (j = index; j < lenOfString; j++) {              if (savedInput1[j] != '>') {                  if (savedInput1[j] == '<') j++;                  break;              }              count = count + 1;          }          if (count > 3) {              printf("There is some error here %s\n", command);              exit(0);          }          while (++j && j < lenOfString) {              if (savedInput1[j] == ' ' || savedInput1[j] == '\t' || savedInput1[j] == '\n')                  continue;              else {                  j--;                  break;              }          }          index = j;          int currIndex = temp[count];          while (j + 1 < lenOfString && savedInput1[j + 1] != ' ' && savedInput1[j + 1] != '>' && savedInput1[j + 1] != '<') {              j++;          }          for (int k = index; k <= j; k++) {              allFiles[count][currIndex][k - index] = savedInput1[k];          }          allFiles[count][currIndex][j + 1 - index] = '\0';          removeWhiteSpaces(allFiles[count][currIndex]);          temp[count]++;          maxLen = max(maxLen, j - index + 1);          index = j;      }  }  int getCommand(char \*input, int len) {      for (int i = 0; i < len; i++) {          if (input[i] == '>' || input[i] == '<') return i;      }      return len;  }  void getFDs(char \*\*allFiles[], int noOfFiles[], int \*fileDescpt[]) {      fileDescpt[0] = (int \*)calloc(noOfFiles[0] + 1, sizeof(int));      fileDescpt[1] = (int \*)calloc(noOfFiles[1] + 1, sizeof(int));      fileDescpt[2] = (int \*)calloc(noOfFiles[2], sizeof(int));      for (int i = 0; i < noOfFiles[0]; i++) {          int fd = open(allFiles[0][i], O\_RDONLY);          if (fd == -1) {              printf("cannot open the file %s\n", allFiles[0][i]);          }          fileDescpt[0][i] = fd;      }      for (int i = 0; i < noOfFiles[1]; i++) {          {              FILE \*f;              f = fopen(allFiles[1][i], "w");              fclose(f);          }          int fd = open(allFiles[1][i], O\_WRONLY | O\_APPEND);          if (fd == -1) {              printf("cannot open the file %s\n", allFiles[1][i]);          }          fileDescpt[1][i] = fd;      }      for (int i = 0; i < noOfFiles[2]; i++) {          {              FILE \*f;              f = fopen(allFiles[2][i], "a");              fclose(f);          }          int fd = open(allFiles[2][i], O\_WRONLY | O\_APPEND);          if (fd == -1) {              printf("cannot open the file %s\n", allFiles[2][i]);          }          fileDescpt[2][i] = fd;      }  }  void getAllFileDescriptors(char \*command, char \*\*actualCommand, char \*\*allFiles[], int noOfFiles[], int \*fileDescpt[]) {      int index = getCommand(command, strlen(command));      char \*fileString = command + index;      (\*actualCommand) = (char \*)calloc(index + 1, sizeof(char));      for (int i = 0; i < index; i++) {          (\*actualCommand)[i] = command[i];      }      (\*actualCommand)[index] = '\0';      segregateFiles(fileString, noOfFiles, allFiles);      getFDs(allFiles, noOfFiles, fileDescpt);  }  void completeCommandExecutionWithRedirection(char \*actualCommand, int \*fileDescpt[], int noOfFiles[], char \*\*allFiles[], char \*arguments[], int index, int n) {      char tempFileOut[] = "/tmp/tempFile-XXXXXX";      int tempFileOutFD = mkstemp(tempFileOut);      char tempFileIn[] = "/tmp/tempFile-XXXXXX";      int tempFileInFD = mkstemp(tempFileIn);      unlink(tempFileIn);      unlink(tempFileOut);      int child = fork();      if (child != 0) {          wait();          for (int i = 1; i <= 2; i++) {              lseek(tempFileOutFD, 0, SEEK\_SET);              char c;              int count = 1;              for (int j = 0; j < noOfFiles[i]; j++) {                  lseek(tempFileOutFD, 0, SEEK\_SET);                  while ((count = read(tempFileOutFD, &c, 1)) > 0)                      write(fileDescpt[i][j], &c, 1);              }              if (i == 1) {                  int j = noOfFiles[i];                  lseek(tempFileOutFD, 0, SEEK\_SET);                  count = 1;                  if (index + 1 == n && noOfFiles[1] + noOfFiles[2] == 0) {                      while ((count = read(tempFileOutFD, &c, 1)) > 0)                          write(STDOUT\_FILENO, &c, 1);                      continue;                  }                  while ((count = read(tempFileOutFD, &c, 1)) > 0)                      write(fileDescpt[i][j], &c, 1);                  c = EOF;                  write(fileDescpt[i][j], &c, 1);              }          }      } else {          int count = 1;          for (int i = 0; i < 1; i++) {              char c;              for (int j = 0; j < noOfFiles[i]; j++) {                  if (fileDescpt[i][j] < 0) continue;                  while ((count = read(fileDescpt[i][j], &c, 1)) > 0)                      write(tempFileInFD, &c, 1);              }          }          if (index == 0) {              if (noOfFiles[0] == 0)                  tempFileInFD = STDIN\_FILENO;          } else {              char c;              while ((count = read(fileDescpt[0][noOfFiles[0]], &c, 1)) == 1 && c != -1) {                  write(tempFileInFD, &c, 1);              }          }          lseek(tempFileInFD, 0, SEEK\_SET);          dup2(tempFileOutFD, STDOUT\_FILENO);          if (!(index == 0 && noOfFiles[0] == 0))              dup2(tempFileInFD, STDIN\_FILENO);          execvp(actualCommand, arguments);          printf("Some error occured while executing %s\n", actualCommand);          exit(0);      }      exit(0);  }  void startCommandExecution(char \*input, int len, bool bgExec) {      char \*\*allWords = NULL;      char \*\*allPipes = NULL;      int pipesLength = 0;      char \*delimiterForCommand = " \t\n";      char \*delimiterForPipe = "|";      splitInputIntoWords(input, len, &allPipes, &pipesLength, delimiterForPipe);      int p[pipesLength + 1][2];      for (int i = 0; i < pipesLength + 1; i++) {          pipe(p[i]);      }      for (int pipeIndex = 0; pipeIndex < pipesLength; pipeIndex++) {          close(p[pipeIndex][1]);          int argumentLength = 0;          char \*command = (char \*)calloc(strlen(allPipes[pipeIndex]), sizeof(char));          strcpy(command, allPipes[pipeIndex]);          char \*\*allFiles[3];          int noOfFiles[3] = {0};          int \*fileDescpt[3] = {0};          char \*actualCommand = "";          getAllFileDescriptors(command, &actualCommand, allFiles, noOfFiles, fileDescpt);          char \*\*allArguments = NULL;          splitInputIntoWords(actualCommand, strlen(actualCommand), &allArguments, &argumentLength, delimiterForCommand);          fileDescpt[0][noOfFiles[0]] = p[pipeIndex][0];          fileDescpt[1][noOfFiles[1]] = p[pipeIndex + 1][1];          int child = fork();          if (child == 0) {              completeCommandExecutionWithRedirection(allArguments[0], fileDescpt, noOfFiles, allFiles, allArguments, pipeIndex, pipesLength);              exit(0);          } else {              wait();          }      }      if (bgExec == true)          printf("done execution %s pid=%d \n", input, getpid(), getppid());      exit(0);  }  int main() {      printWelcome();      size\_t MAX\_LEN = 0;      char \*input = NULL;      char \*\*allWords = NULL;      char \*\*allPipes = NULL;      int pipesLength = 0;      int argumentLength = 0;      char \*pwd = (char \*)calloc(PATH\_MAX, sizeof(char));      char \*absolutePath = (char \*)calloc(PATH\_MAX, sizeof(char));      char \*delimiterForCommand = " \t\n";      char \*delimiterForPipe = "|";      while (1) {          printf("cmd >", pwd);          if (input)              free(input);          MAX\_LEN = 0;          int len = getline(&input, &MAX\_LEN, stdin);          if (strcmp(input, "exit\n") == 0) {              break;          }          if (strcmp(input, "\n") == 0) {              continue;          }          bool bgExec = false;          for (int i = len - 1; i >= 0; i--) {              if (input[i] == ' ' || input[i] == '\t' || input[i] == '\n')                  continue;              else {                  if (input[i] == '&') {                      input[i] = '\0';                      bgExec = true;                  }                  break;              }          }          int startChild = fork();          if (startChild == 0) {              if (bgExec)                  printf("Started background execution pid=%d\n", getpid());              startCommandExecution(input, len, bgExec);              exit(0);          } else {              if (!bgExec) {                  wait();              }          }      }  } |

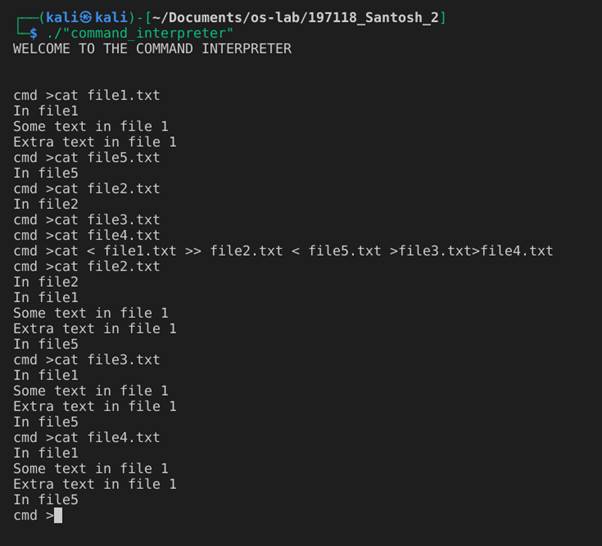
Command execution:



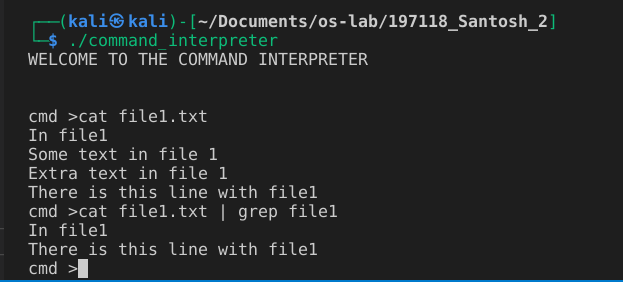
Input Redirection:



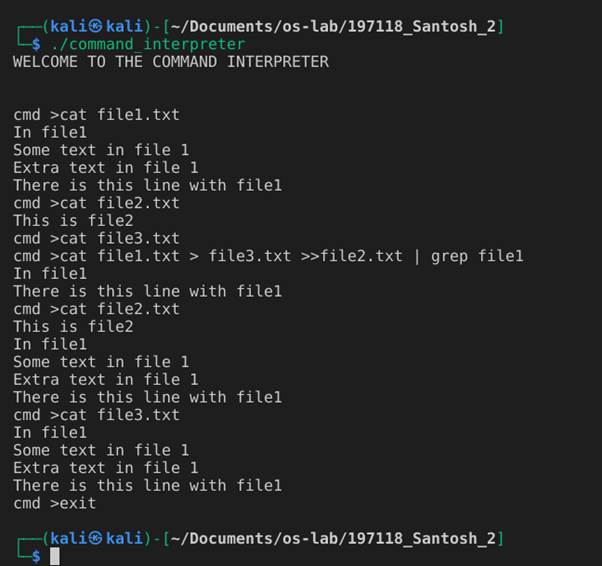
Multiple IO redirections:



Piping example:



Multiple IO redirections with Piping:



With background executio:

