**Some Example Programs for Project 1**

In the following some example code segments are presented with some explanatory details. You may find them useful for project1 if you are not strong in C/C++ programming. You will need to understand them, modify them, and put them together appropriately to come up with an integrated solution for the project.

It is not mandatory to use these code examples in your project. You can explore online to find more examples to learn and use available code samples as suitable for your project. Don’t forget to list the sources in the readme file for the project.

1. **Tokenizing a String**

You need to know how to use a tokenizer function in a program to separate tokens or words in a command or string using a delimiter such as a space or any other special character. You can use **strtok()** function to tokenize a given string. Go online to learn about **strtok()** function.

The tokens or words that you obtain are to be used as arguments in an **exec** function to run a command.

If you are good in programming, you can write your own tokenizer function and use it for Project 1.

|  |
| --- |
| **Example 1. Tokenizer Using strtok() Function** |
| // Extracting string tokens using strtok()  #include <stdio.h>  #include <string.h>    int main()  {  char str[200];  int i = 0;  if(fgets(str, 200, stdin)==NULL) // The statement reads a line  {  printf("Error\n");  return 0;  }    str[strlen(str) - 1] = '\0'; // It replaces newline ch with null  // Returns first token with delimiter space  char\* token = strtok(str, " ");    // Keep extracting tokens  while (token != NULL) {  printf("%s\n", token); // shows each token at a time  token = strtok(NULL, " ");  }  return 0;  } |

In example 1, the program takes a line of input from the standard input device stdin, which is the keyboard, by using the fgets() function. The function fgets() places the line of input (including the new line character ‘\n’) in the character array str which can hold at most 200 characters as declared in the code. To make the character array str a C string, the newline character at the end is replaced by the null character ‘\0’. This is needed to use the tokenizer function strtok(). In this example, the tokenizer function strtok() uses space as a delimiter to extract tokens. The function strtok() returns the character pointer to the first token if the function call is successful. The while loop is used in the example to extract the remaining tokens one at a time. For the demonstration purpose the printf() function prints each token at a time. Note that a token returned by the strtok() function is a null terminated string. Also, note that space is used as the delimiter in this example. It is possible to use any other character or symbol as a delimiter. Copy, paste, compile, and run the example code for your understanding.

1. **Creating a Process**

Typically, a single program runs as a single process in a Unix environment. However, to take advantage of parallelism or to fulfill some other computational needs you can create many processes as needed for your application. The fork() function can be used to create a child process. The child process is essentially a separate image of the parent process and gets executed separately and concurrently. As far as programming is concerned, all objects, variables, code blocks, etc. are copied for the child process. Following the fork() function call, both the child process and the parent process continue to run. Interestingly, the same fork() function returns two integer values, a value of 0 to the child process and a non-zero value to the parent process (this non-zero value is the ID of the child process). A programmer may use the return value of the fork() function to selectively execute different code blocks for the parent process and the child process. It is to be noted that the fork() function returns a negative number if it fails.

In example 2, immediately after the fork() function call, two processes will be running and each of the processes will execute its own copy of the if statement in its own code area. The code block immediately after the if statement will be executed in the child process since the condition pid == 0 is true, but the block immediately after the else statement will be executed in the parent process since the condition pid == 0 is false. Test the program to develop and solidify your understanding.

|  |
| --- |
| **Example 2. Using fork() Function** |
| #include <stdio.h>  #include <unistd.h>  int main(int argc, char \*argv[])  {  int pid;  pid = getpid();  printf("Parent process ID: %d\n", pid);  pid = fork();  if (pid == 0) // Child process executes this block  {  printf("Return value in child process: %d\n", pid);  pid = getpid();  printf("Process ID reported by child process: %d\n", pid);  }  else // Parent process executes this block  {  printf("Return value in parent process: %d\n", pid);  pid = getpid();  printf("Process ID reported by parent process: %d\n", pid);  }  return 0;  } |

1. **Executing a Command or an Application**

There are several types of the exec function that can be used to execute a command or an application from a C program. In this tutorial, we will focus on the use of the execv() function to execute a command or an application. You need to prepare the arguments of the function with C strings before you call the function in your program. It takes two arguments. The first argument is the path to the directory in which command or the executable resides, and the second argument is a vector that holds the command (or app) name and the options as successive elements followed by a NULL pointer as the last element in the vector. The path and all elements of the vector are C-Strings. In example 3, we run the ls -l command by using the execv() function. Notice how the program allocates memory areas for three elements of the vector argv and fills them with the command ls, an option -l, and the NULL pointer respectively. Notice also how the path information is built in the allocated folder string.

|  |
| --- |
| **Example 3: Using execv() function** |
| #include <string.h>  #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  int main(void)  {  char\* argv[10];  char\* folder;  folder = malloc(200); // Allocate 20 character positions  argv[0] = malloc (50);  argv[1] = malloc(50);;  argv[2] = malloc(50);;    strcpy(argv[0], "ls"); // Command goes in argv[0]  strcpy(argv[1], "-l"); // option  argv[2] = (char \*) 0; // last argument must be NULL  folder = strcpy(folder, "/bin/");  folder =strcat(folder, argv[0]); // Here the folder is "bin/ls"  execv(folder, argv); // call to the function; execute and exit    printf("Unknown command\n"); // this statement will not be executed if excecv() is successful  return 0;  } |

When any exec function runs successfully, it also exits the process in which it runs. Only, in the case if there is an error in the exec function, the process continues. For your project, your execv() function must be called in a child process so that the parent process will still continue when the child process exits after the successful execution of the command.

|  |
| --- |
| Example 4: Using execv() and fork() functions together to run a command in a child process |
| // Write your name here  #include <string.h>  #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  int main(void)  {  char\* argv[10];  char\* folder;  int process\_id;  int status;  // The following lines of code is to prepare arguments for execv() function  folder = malloc(200); // Allocate 20 character positions  argv[0] = malloc (50);  argv[1] = malloc(50);  argv[2] = malloc(50);  strcpy(argv[0], "ls"); // Command goes in argv[0]  strcpy(argv[1], "-l"); // option  argv[2] = (char \*) 0; // last argument must be NULL  //argv[1] = (char \*) 0;  folder = strcpy(folder, "/bin/");  folder =strcat(folder, argv[0]); // Here the folder is â€œbin/lsâ€  // All arguments are ready to call execv() function at this point  // Create a child process  process\_id = fork();  if (process\_id == 0) // in child process  {  execv(folder, argv); // call to the function in child process; execute and exit from the child process  printf("Unknown command\n"); // this statement will be executed if excecv() fails  }  else // in parent process  {  wait(&status); // parent process waits  printf("Parent process continues\n");  }    return 0;  } |

Example 4 shows how to run a command using execv() function in a child process. Notice that any successful execution of the execv() function terminates the child process. Any failure in execution of the execv() function makes the child process to continue in the next statement. However, the parent process always continues.

In examples 3 and 4, the path, the command, and the option are hardcoded in the program. Here is a question: How can you receive the command as an input line from the keyboard and fill in the arguments of the execv() function? The answer is using a tokenizer to do the job.

1. **Maintaining History of Commands**

For project 1, you need to know how to use or develop a data structure to maintain history of at most 10 recent commands entered. You may use a circular array of 10 strings. You many need to maintain the command strings, display them on the screen, retrieve a command from the history data structure, and on.

Example 5 shows a simple class hist that can be used for your purpose as a circular buffer for 10 strings. As coded, it can maintain up to ten C strings, each of size of at most 100 characters including the NULL character. You can use the add() method to append a string in a hist object, the get() method to retrieve a string from the object, and the display() method to show all strings on the screen from the object. This class can be used in conjunction with the strtok() function as a tokenizer which you can call with a retrieved string.

|  |
| --- |
| Example 5. hist Class |
| #include <iostream>  #include <cstring>  #include <stdio.h>  using namespace std;  class hist  {  private:  char data[10][100]; // It can hold at most 10 C-strings with maximum length of 100 characters  int count;  public:  hist();  void add(char[]);  void get(int, char[]);  void display();  };  hist::hist()  {  count = 0;  for (int i = 0; i < 10; ++ i)  strcpy(data[i], "");  }  void hist::add(char x[])  {  int i;  if (count < 10)  {  strcpy(data[count], x);  ++ count;  }  else // shift if it is full  {  for (i = 0; i < 9; ++ i)  strcpy(data[i], data[i+1]);  strcpy(data[9],x);  }  return;  }  void hist::get(int x, char y[])  {  if (x <= count)  strcpy(y, data[x - 1]);  else  strcpy (y, "");  return;  }  void hist::display()  {  for (int i = 0; i < count; ++ i)  cout << data[i] << endl;  return;  }  // The following code is to test the hist class and its methods  int main()  {  hist commands; // Declare the hist data structure  int n;  char x[100];  fgets(x, 100, stdin); // Get at most 100 character from the keyboard  x[strlen(x)-1]='\0'; // Append null character at the end  commands.add(x); // Include the string in the hist object    // Get another string and save in hist  fgets(x, 100, stdin);  x[strlen(x)-1]='\0';  commands.add(x);    commands.display(); // Show all strings stored in hist    commands.get(1, x); // Retrieve a string at index 1 from hist object  cout << x << endl; // Display the retrieved string  return 0;  } |

In addition to the tokenizer and the data structure discussed above, you may find a more suitable tokenizer and a more suitable data structure. You are welcome to explore and use them as you find them useful for your project.