UMass Boston CS 444
Project 2
Posted Tuesday, October 28, 2025
Due Monday, November 10, 2025 at 11:59pm

J.H.DeBlois

1 Introduction for bsh.c

In this project, we develop a "shell" program, a C program called bsh, which stands for b-shell or Boston shell. It works like the classic sh and bash. Your program bsh will include code to handle five new commands. The five are required and are described below. Optionally, your program bsh will include code to handle four more commands for extra credit, also described below.

For proj2, you are given starter code to add to. You are also given two demo programs. In this write-up, command line is abbreviated as CL.

The program bsh accepts commands in either of the two following syntax formats:

```
command [arg1 ... argn] [< filename] [> filename]   //CL arguments with I/O redirection
command [arg1 ... argn] | command [arg1 ... argn]   //CL arguments with a pipe between two commands
```

First, create a directory in your cs444 folder on the CS server:

```
pe15> cd ~/cs444
pe15> mkdir proj2
```

Next, copy four files from the instructor's directory to the your project directory:

```
pe15> cp /home/hdeblois/cs444/proj2/* ~/cs444/proj2
```

The files are the following:

```
-rw-r--r-- 1 hdeblois 2900 Oct 26 20:58 bsh.c
-rw-r--r-- 1 hdeblois  273 Oct 26 20:58 envDemo.c
-rw-r--r-- 1 hdeblois   74 Oct 26 20:58 Makefile
-rw-r--r-- 1 hdeblois 1580 Oct 26 20:58 pipeDemo.c
```

The supplied Makefile makes bsh.c. For the demo programs, you can compile with gcc. Then run a.out and copy it to the appropriate executable name, envDemo or pipeDemo. To give you a quick idea on how to start, bsh.c already does the exit command. The bsh.c program also has stub versions of the basic commands, but not pwd. Compile and try these commands:

```
prompt>./bsh
bsh>env
bsh>setenv
bsh>unsetenv
bsh>cd
bsh>history
bsh>pwd
      no such command (pwd)
bsh>exit
prompt>
```

We make a simplification to start by assuming that the user of the bsh program will enter commands in a clean way such that all parts are separated by spaces. Here is an example:

```
cat in.txt | wc > out.txt
```

For extra credit, as explained below, you may modify your bsh so it can handle commands typed as follows:

```
cat in.txt|wc>out.txt
```

2 Background

In this section, we discuss the required commands, environment variables in general and the string.h function strsep() that you need to use to make the bsh program able to separate parts of the command string.

The bsh program is designed to have six built-in commands including exit which is already implemented. You are required to write C code to implement the other five. For sure, do not fork a new process and then invoke existing bash commands to accomplish the tasks.

The six built-in commands and their syntax or output format are:

1. exit – exits bsh.
2. env – lists the environment variables and their values.

   The output format for env is as follows (one var=value per line);

   SHELL=/bin/bash

   EDITOR=emacs

   PWD=/home/hdeblois/cs444/proj2

   LOGNAME=hdeblois
3. setenv – sets the value of an environment variable (new or existing).

   The syntax for setenv is: setenv variable value
4. unsetenv – removes a variable from the environment.

   The syntax for unsetenv is: unsetenv variable

5. cd – changes the current working directory and updates the environment variable PWD.
6. history – lists the last 500 commands the user has entered.

Environment variables are typically inherited from the parent shell. When an executable such as bsh starts running, it receives three input parameters. Look at the main in envDemo.c to see this:

```
int main(int argc, char *argv[], char *envp[]){
}
```

The first two parameters should be familiar to you: argc is the number of CL arguments, and argv is an array of string pointers to the arguments. The third parameter is probably new to you. Just like argv, it is an array of string pointers where each string is formatted on a line including the parameter in CAPS, an equals sign and the value. Here is an example:

```
SHELL=/bin/bash
```

The main difference between argv and envp is:
You know there are argc arguments in argv[]. They are stored in argv[0], argv[1], . . ., argv[argc-1].
You don't know the number of environment variables right away. If there are k such variables, they are stored in envp[0], envp[1], . . ., envp[k-1]. The only way for you to know that you have reached k is by testing for envp[k] == NULL.

Throughout this project, you often need to separate a string into several tokens. The function strsep() found in string.h is handy for this purpose. It separates a string by delimiters of your choice.

For example, here is code that uses strsep():

```
char tmpStr[1024], *myPath, *justPATH;
strcpy(tmpStr, "PATH=/bin:/usr/bin:/usr/local/bin");
myPath = tmpStr;
justPATH = strsep(&myPath, "=");
```

The following summarizes what happened above.
The variable tmpStr is unchanged — it still points at the same address.
justPATH has the same value as tmpStr — it points at the same address.
strsep() replaced the equals sign at tmpStr[4] with the end-of-string

```
char '\0'
```

so if you print the string at justPATH, you get the PATH literally, since strsep() separated it from the equals sign and what went before.
myPath points at tmpStr[5], just beyond the original equals sign — if you print the string at myPath, you get /bin:/usr/bin:/usr/local/bin.
Last but not least, strsep() is destructive — the copy in tmpStr is altered. This is why we used strcpy() first. You want to tokenize the copy — the original PATH value should be kept intact.
P.S. In the section on Linux commands below, you can loop through the paths by:

```
strsep(&myPath, ":").
```

With this background information in hand, the individual tasks can now be described. First, we describe the required tasks. Then we describe the extra credit tasks.

3 Required Tasks

3.1 Implement three built-in commands that relate to environment variables

To implement env, setenv, and unsetenv, you need to do the following:
• At the beginning of bsh, loop through envp and make a copy of the environment variables to the memory space of your code. If you don't make a copy and later try to change the values directly in envp, disasters — segfaults — will strike. To store the environment variables, it is easier to use an array than a linked list. You may assume that there are no more than 64 variables — you will see that bsh.c already has:

```
#define MAXENV 64
```

• When the users of bsh — you and the grader – try to setenv, you search your copy of the environment variables. If it is an existing variable, you should malloc() space for a new string, save the new value, and free() the old string. If you don't malloc() new space, the existing space may not be long enough to accommodate the new value, and segfaults will strike. If the environment variable is new one, you don't need to free() the existing space, and you can just malloc() a new space to save it.
• When the users try to unsetenv, you remove the variable from the list. Don't forget to free() the memory, or else there will be memory leak.

3.2 Implement Change Directory

You use the system call chdir() to change the working directory of bsh. There are several cases of where the user wants to go:
• If the user enters just cd or cd , you chdir() to the user's home directory, which is stored in the environment variable HOME.
• If the user enters cd someDir, you chdir() to someDir.
• You should allow the user to enter a relative path that begins with the dot or dot-dot notations,

```
cd ./someDir or cd ../someDir.
```

After calling chdir(), don't forget to set the value of the environment variable PWD accordingly. When you save a new value to PWD, it is safer to malloc() a new string with enough space. Oth- erwise, if the new PWD is longer than the old PWD, segfaults may strike. Don't forget to free() the old memory, or else there will be memory leak. You may find the system call getcwd() useful here.

3.3 Implement History

The Linux command history lists chronologically the last 500 commands that a user has entered. Implement this feature for the bsh program. In bsh.c, there is:

```
#define MAXLINE 1024
```

which limits a command line to no more than 1024 bytes. Therefore, use an array of 500 pointers, each pointing to a string of 1,024 bytes, so that you don't need to malloc() and free() all the time.
However, before the user has entered 500 commands, you need a way to know which history slots are valid and which slots are yet to be filled.

4 Extra Credit Tasks

4.1 Implement Finding Other Commands in Directories of PATH

An environment variable called PATH has a list of directories that contain Linux executables. When a command entered by the user is not one of the built-in commands, the bsh program should check to see if the command exists in one of the directories in PATH.

You need to do the following:
1) Iterate through the directories listed in PATH.
2) Use strsep() to separate the paths. However, you should make a copy of PATH=... and apply strsep() to the copy, because strsep() is destructive. You must keep the original copy of PATH=... intact so that you can use it again.
3) Append the user command to the end of a path to make an absolute path. For example, if the user command is ls, and you have extracted a path /usr/bin from PATH, then you concatenate them to create the absolute path /usr/bin/ls.
4)Use the system call access() to see if the absolute path is a valid executable.
5) If access() says it is indeed an executable, you run it on behalf of the user as follows:
   a) Use fork() to generate a child process
   b) Make the parent process wait for the child
   c) In the child process, call execv() with the appropriate parameters to run the executable. See bsh.c for details.
6) If the command does not appear anywhere in the paths, an error message should be printed. To implement the full functionality of bsh, you may need several system calls: fork(), wait(), waitpid(), execv(), chdir(), access(). You can find their manual pages by the man command in Linux:

```
pe15> man fork
```

or you can just google the term man fork.

4.2 Implement I/O Redirection

The bsh program needs to be able to handle redirection of stdin and stdout.
The following commands are examples of I/O redirection:

```
bsh> ls > tmp.txt           //Redirects stdout to the file tmp.txt
bsh> wc < tmp.txt           //Redirects stdin from the file tmp.txt
bsh> wc < in.txt > out.txt  //Redirects stdin from in.txt, stdout to out.txt
```

When a child process is created using the system call fork(), it gets a copy of its parent's file descriptor table. Included in this table are the file descriptors for stdin (fd 0) and stdout (fd 1). Each of these can be redirected by closing them and then creating a new file descriptor in their place using the system calls open(), close(), and dup(). For instance, to redirect output from stdout to the file tmp.txt, you do the following:

```
fid = open("tmp.txt", O_WRONLY | O_CREAT);
close(1);       //closes stdout
dup(fid);       //fid is now associated with fd 1
close(fid);     //fid is no longer needed; use fd 1 instead
```

The system call dup() duplicates fid to the first available entry in the file descriptor table, in this case, fd 1, because it was just closed at the previous line.

4.3 Implement Parsing User Commands

If the user enters commands in a clean way that all parts are separated by spaces, then the function strsep() is all you need to tokenize the CL. For example:

```
cat in.txt | wc > out.txt
```

means the function strsep() is all you need. But for this task, you implement a way to allow the user or grader to enter a command like this:

```
cat in.txt|wc>out.txt
```

which requires more design and additional code.

4.4 Implement Pipe

A pipe is a one-way communication channel between two processes. One process writes to one end of the pipe, and the other process reads from the other end. The process that reads from the pipe should know it is time to exit when it reads an EOF on its input. Pipes are created using the system call pipe().

Adapt the code in pipeDemo.c to bsh.c. You are required to implement only one pipe, which connects two commands. Do not worry about multiple pipes connecting more than two commands.

5 Grading Rubric

Be sure to put all files you are submitting into your proj2 directory. Collection will be incremental, so be sure to upload your files to the server often. Before working, copy your prior work to a name indicating the version by appending the date. Then continue to develop your code in your C file. Be sure to cite any online sources you use in your readMe.txt. If you copy code from any of these sources or use code generated from chatgpt, you need to include a comment to identify the source or prompt used and date, indicate "copied from" or "copied and modified" or "generated by" and delineate the lines of the code that was brought in by writing "START" at the beginning of the section and "END" in a comment at the end of the section. We may call you in to explain your code. If you copy excessively, MOSS may give you too high a similarity number.

There are some limitations on extra credit points earned. You may apply extra credit points from proj2 to your proj2 grade, but only if you complete the required five commands in addition to exit. This is for earning up to 100 points total, but not beyond. If you earn extra credit past that, it can be applied to your other C code project(s) up to earning 89 points total. The 89 limit is there so the grade A is reserved for those who earn it across all assignments without using extra credit.

Here are the point values:

a) for required tasks:
(20 points) Write a readMe.txt. Make an entry each time you work to explain what you have done. Identify parts that were hard for you. Do not stay stuck.
(30 points) The commands env, setenv, unsetenv work.
(30 points) The command cd works.
(20 points) The command history works.

b) and for extra credit tasks:

(15 points) Finding and running Linux commands works.
(10 points) I/O redirection works.
(15 points) Parsing user commands without requiring that all parts are separated by spaces works.
(10 points) Combining commands via a single pipe works.