CSCI3150 – Tutorial 5

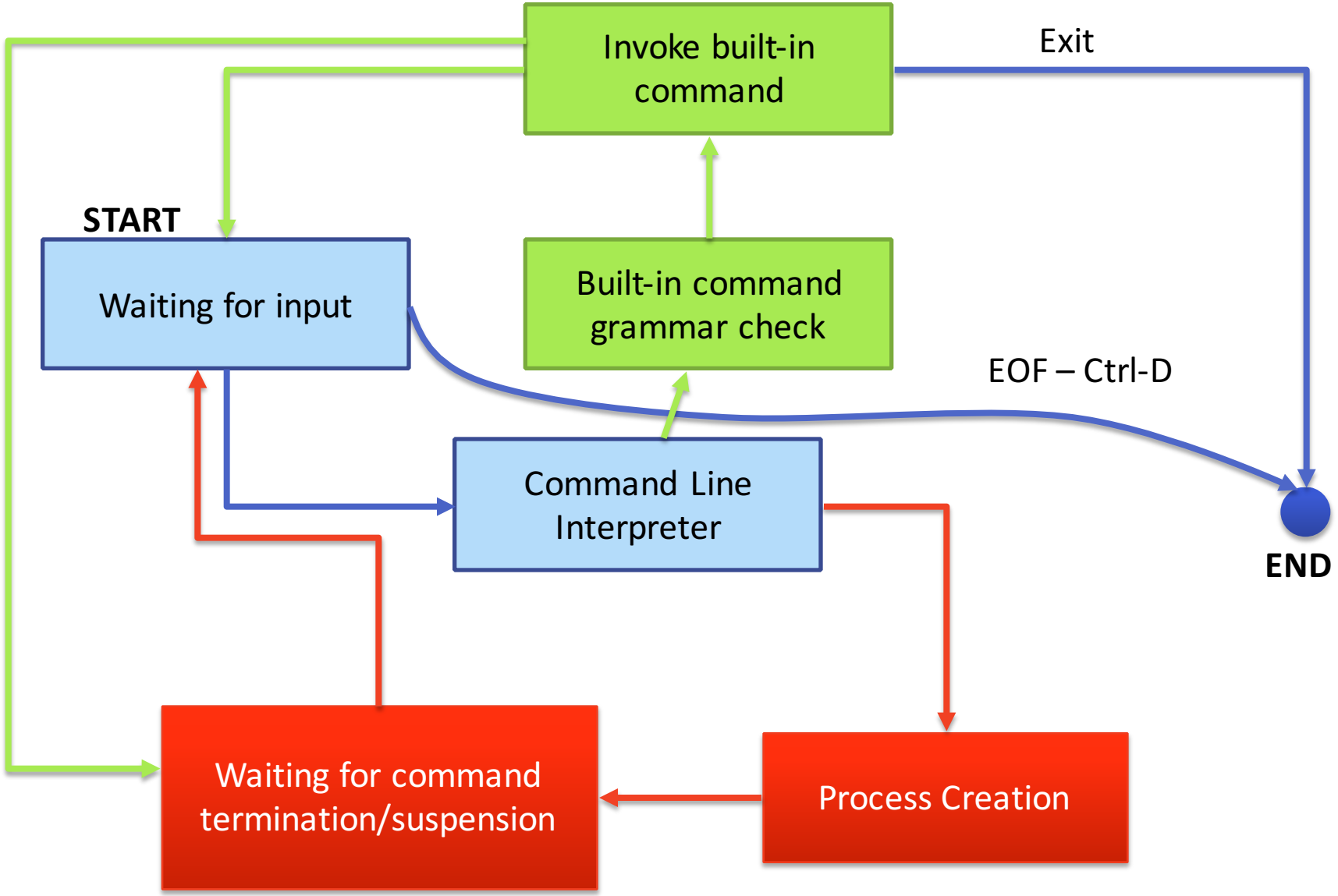
ASSIGNMENT 1, PHASE I REVISITED

Calvin Kam <hckam@cse.cuhk.edu.hk>

Phase 1 – Shell

You will need to write a workable "shell" that can

1. Accept user inputs
2. Analyze the input command line and distinguish if it is built-in commands or others
3. If it is built-in commands:
   1. Change working directories, or exit the program
4. If it is other commands:
   1. Set the search path
   2. Expand the wildcard expression
   3. Fork a new process, set the signal handlers and execute.
   4. Wait for the termination of the child.


Invoke built-in command
Exit
START
Waiting for input
Built-in command grammar check
EOF – Ctrl-D
Command Line Interpreter
END
Waiting for command termination/suspension
Process Creation

Phase 1 – Shell

You will need to write a workable "shell" that can

1. Get user inputs
2. Analyze the input command line and distinguish if it is built-in commands or others
3. If it is built-in commands:
   1. Change working directories, or exit the program
4. If it is other commands:
   1. Set the search path
   2. Expand the wildcard expression
   3. Fork a new process, set the signal handlers, and and execute.
   4. Wait for the termination of the child.

Get the User Input

<stdio.h>

```
[3150 Shell:/home/hckam]$ ls -al
```

- Get the whole input including spaces…. “ls -al“ whether than “ls” alone.
- `fgets()` is recommended, as it is easy to use and get all the things you need.

```c
#include <stdio.h>
#include <string.h>

int main(int argc,char *argv[]){
    char buf[255];
    printf("[I am Shell]$:");
    fgets(buf,255,stdin);
    buf[strlen(buf)-1] = '\0';
    printf("%s\n",buf);
    return 0;
}
```

Get the User Input

If the user presses Ctrl –D (End of file, NOT SIGNAL!!!), then your shell should terminate.

We can detect the EOF by checking the return value of fget()!

```
char buf[255];

if(fgets(buf,255,stdin) != NULL)
        printf("[%s]\n",buf);

printf("\nThe program has terminated!\n");
return 0;
```

Phase 1 – Shell

You will need to write a workable "shell" that can

1. Get user inputs
2. Analyze the input command line and distinguish if it is built-in commands or others
3. If it is built-in commands:
   1. Change working directories, or exit the program
4. If it is other commands:
   1. Set the search path
   2. Expand the wildcard expression
   3. Fork a new process, set the signal handlers, and and execute.
   4. Wait for the termination of the child.

Tokenize the input

<string.h>

After you get the input, the string is “chopped” into words (tokens) in order to be examined (for example, check whether it is a built-in command) and build the argument list.

A useful string function - `char *strtok(char *str, const char *delim);`

```
#include <string.h> //Needed by strtok()
char buf[] = "Hello World tywong sosad";
char *token = strtok(buf," ");
while(token != NULL)
{
printf("%s\n",token);
token = strtok(NULL," ");
}
```

Built-in Commands

In this assignment, we only have two built-in commands!

For the kind of **built-in commands**, you need to write your own code.

1. Change Directory (cd)

2. Exit the shell (exit)
   - Break the while loop, and we are done :)

Phase 1 – Shell

You will need to write a workable "shell" that can

1. Get user inputs
2. Analyze the input command line and distinguish if it is built-in commands or others
3. If it is built-in commands:
   1. Change working directories, or exit the program
4. If it is other commands:
   1. Set the search path
   2. Expand the wildcard expression
   3. Fork a new process, set the signal handlers, and and execute.
   4. Wait for the termination of the child.

Get Current Directory

<unistd.h>
<limits.h>

In the prompt, we need to know the current working directory.

Initially the path is where the program is located.

The useful function is - `char *getcwd(char *buf, size_t size);`

PATH_MAX is the maximum characters allowed in the pathname.

```c
#include <stdio.h>
#include <limits.h> // Needed by PATH_MAX
#include <unistd.h> // Needed by getcwd()

int main(int argc,char *argv[]){
     char cwd[PATH_MAX+1];
     if(getcwd(cwd,PATH_MAX+1) != NULL){
     printf("Current Working Dir: %s\n",cwd);
     }
     else{
     printf("Error Occured!\n");
     }
     return 0;
}
```

Change the working directory [cd]

<unistd.h>

It is a **built-in** command and it changes the current working directory.

The useful system call is `int chdir(const char *path);`

```
char buf[PATH_MAX+1];
Char input[255];
if(getcwd(buf,PATH_MAX+1) != NULL){
    printf("Now it is %s\n",buf);
    printf("Where do you want to go?:");
    fgets(input,255,stdin);
    input[strlen(input)-1] = '\0';
if(chdir(input) != -1){
    getcwd(buf,PATH_MAX+1);
    printf("Now it is %s\n",buf);}
else{
        printf("Cannot Change Directory\n");
}
```

Remember to remove the newline character!

Phase 1 – Shell

You will need to write a workable "shell" that can

1. Get user inputs
2. Analyze the input command line and distinguish if it is built-in commands or others
3. If it is built-in commands:
   1. Change working directories, or exit the program
4. If it is other commands:
   1. Set the search path
   2. Expand the wildcard expression
   3. Fork a new process, set the signal handlers, and and execute.
   4. Wait for the termination of the child.

Set the search path

REQUIREMENT: (1) The shell allows file names (2)Search for the program

The sequence of the search path is determined by the environment variable **$PATH.**

/bin → /usr/bin → . (current directory)

To do : set the environment variable to our desired search path

setenv()

<stdlib.h>

int setenv(const char *_name_, const char *_value_, int _overwrite_)

This function can change the environment variable in the system temporarily until the termination of the process.

```
char *command1[] = {"shutdown",NULL};
printf("Running shutdown.. it is in /sbin :P \n\n");
setenv("PATH","/bin:/usr/bin:.",1);
execvp(*command1,command1);

if(errno == ENOENT)
        // Print No command found message
else
        // Print unknown error message
return 0;
```

Now the shell only searches the specified path 😺

Phase 1 – Shell

You will need to write a workable "shell" that can

1. Get user inputs
2. Analyze the input command line and distinguish if it is built-in commands or others
3. If it is built-in commands:
   1. Change working directories, or exit the program
4. If it is other commands:
   1. Set the search path
   2. Expand the wildcard expression
   3. Fork a new process, set the signal handlers, and and execute.
   4. Wait for the termination of the child.

Wildcard expansion

It is the shell's responsibility to expand to wildcard character in command lines.

Requirements:

1. Expands the wildcard expansion into series of tokens, and they are in lexicographical order.
2. If the file is not found, keep the expression unchanged.

glob()

To find the filenames with the given pattern, you can use glob().

glob() function utilizes a special structure called **glob_t**

```
typedef struct {
    size_t   gl_pathc;    /* Count of paths matched so far  */
    char   **gl_pathv;    /* List of matched pathnames.  */
    size_t   gl_offs;     /* Slots to reserve in gl_pathv.  */
} glob_t;
```

Reserve 1 space for the command name.

glob()

```
#include <glob.h>

int glob(const char *pattern, int flags,
         int (*errfunc) (const char *epath, int eerrno),
         glob_t *pglob);
```

1. lexicographical order :
   - Be default it is sorted, no need to do extra things.
2. Place the command before the expanded tokens.
   - Use GLOB_DOOFFS
3. Returns the original tokens upon no matches
   - Use GLOB_NOCHECK
4. Call glob() on each token
   - Remember to add GLOB_APPEND on the 2nd and onward call.

glob()

```
1  #include <stdio.h>
2  #include <glob.h>
3  #include <unistd.h>
4
5  int main(int argc,char *argv[])
6  {
7          glob_t globbuf;
8
9          globbuf.gl_offs = 1;
10
11         glob("*.c", GLOB_DOOFFS | GLOB_NOCHECK, NULL, &globbuf);
12         glob("*.txt", GLOB_DOOFFS | GLOB_NOCHECK | GLOB_APPEND, NULL, &globbuf);
13         glob("*.java", GLOB_DOOFFS | GLOB_NOCHECK | GLOB_APPEND, NULL, &globbuf);
14         globbuf.gl_pathv[0] = "ls";
15
16         execvp(globbuf.gl_pathv[0],globbuf.gl_pathv);


           return 0;
}
```

Put the command name at front

Phase 1 – Shell

You will need to write a workable "shell" that can

1. Get user inputs
2. Analyze the input command line and distinguish if it is built-in commands or others
3. If it is built-in commands:
   1. Change working directories, or exit the program
4. If it is other commands:
   1. Set the search path
   2. Expand the wildcard expression
   3. Fork a new process, set the signal handlers, and and execute.
   4. Wait for the termination of the child.

Signals

To emulate the real "shell", your shell should handle some signals properly.

According to the specification, the following signals should be ignored.

Signals	Handling routines
SIGINT (Ctrl + C)	Ignore
SIGKILL (Default for kill)	Ignore
SIGQUIT (Ctrl + \)	Ignore
SIGTSTP (Ctrl + Z)	Ignore

However, your children should **restore** the above signal handling routines. **REMEMBER!!!!!!!!**

Signals

```
<stdio.h>
<signal.h>
<unistd.h>
<sys/types.h>
<sys/wait.h>
```

```
1  int main(int argc,char *argv[]) {
2            signal(SIGINT,SIG_IGN);
3            signal(SIGQUIT,SIG_IGN);
4            signal(SIGTERM,SIG_IGN);
5            signal(SIGTSTP,SIG_IGN);
6            if(!fork()) {
7                     signal(SIGINT,SIG_DFL);
8                     signal(SIGQUIT,SIG_DFL);
9                     signal(SIGTERM,SIG_DFL);
10                    signal(SIGTSTP,SIG_DFL);
11                    printf("[%d] I am Child..\n",getpid());
12                    while(1) {}
13           }
14           else{
15                    wait(NULL);
16                    printf("[%d] I am super parent, kill me if you can\n",getpid());
17                    while(1) {}
18           }
19           return 0;
20 }
```

Remember

fork() + exec()

<unistd.h>

We know that we need to fork a new process to use the system call exec().

```c
#include <stdio.h>
#include <unistd.h> // Needed By fork(),sleep()
int main(int argc,char *argv[]){
        while(1)
        {
                printf("\nPress Enter to execute ls...");
                while(getchar() != '\n');
                if(!fork()) {
                        char *arglist[] = {"ls",NULL};
                        execvp(*arglist,arglist);
                }
                else{
                        sleep(1);
                }
        }
        return 0;
}
```

Is it okay? 👻

How to handle the child properly

<unistd.h>
<sys/wait.h>
<sys/types.h>

The shell should wait until the child dies and handle the event.

The useful system call is **pid_t wait(int *_status_);**

```
while(1) {
        printf("\nPress Enter to execute ls...");
        while(getchar() != '\n');
        pid_t child_pid;
        if(!(child_pid = fork())) {
                char *arglist[] = {"ls",NULL};
                execvp(*arglist,arglist);
}
                else{
                        wait(NULL);
                }
        }
```

Last Reminder

```
if(!fork()) {
        char *arg[] = {"Hello",NULL};
        execvp(*arg,arg);
        printf("Error!\n");
}
else {
        wait(NULL);
}
```

In case of error in exec call(), exec will **return** and the program continues execution. You need to handle that.

Enjoy :)

Phase 1 **DEADLINE:** **23:59, 2016 Mar 4 (Fri)**

INDIVIDUAL work. ;)