UNIX Shell Project

Operating Systems

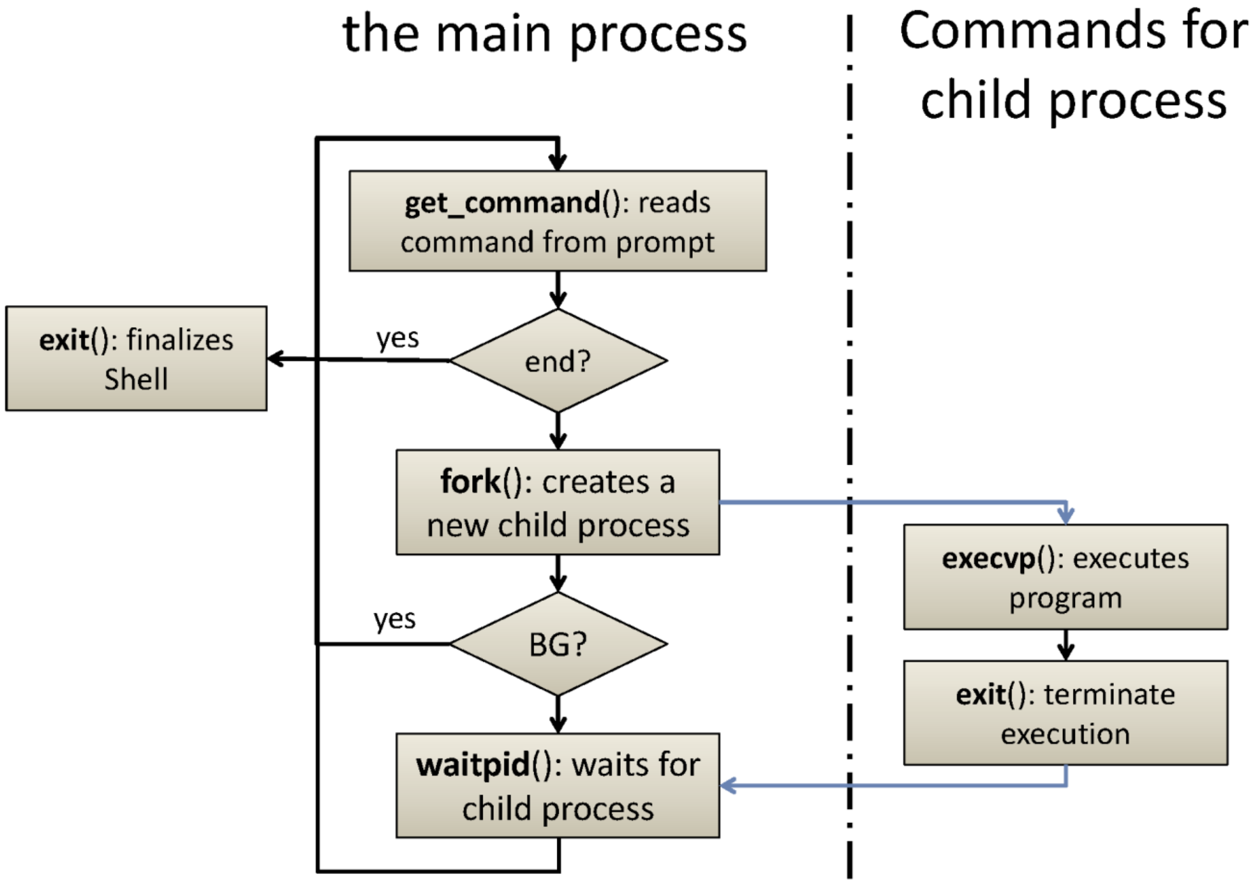
Group B, English

Dept. Computer Architecture - UMA

The aim of this lab is to build a Shell that provides a command-line interpreter. The idea is to achieve user-system interaction through the new Shell. The user will use it to run OS-commands and his/her own commands.

The Shell should read a command line typed by the user and create a new process to execute that command. To implement this program you will need to use these system-calls: **fork**: creates a child process, **execvp**: allows the child process to execute a program different from the parent program, **waitpid**: waits for a child to finalize and **exit**: terminates a process. Please, make use of the Linux **man**ual to learn how these and other functions work.

A simplified scheme of how the Shell works:


the main process
Commands for child process
get_command(): reads command from prompt
end?
yes
exit(): finalizes Shell
fork(): creates a new child process
BG?
yes
waitpid(): waits for child process
execvp(): executes program
exit(): terminate execution

To build your Shell you may use the C source codes available in the Campus Virtual:

- "`Shell_project.c`" This code can be used as a starting point to implement the shell with task control following the concepts described in the appendix at the end of this document. Please read the appendix first.

- You may use the functions available in the module "`job_control.c`"

The most relevant functions and macros of module "`job_control.c`" are:

Description of "`job_control`" module

get_command

```
void get_command(char inputBuffer[], int size, char *args[], int *background);
```

This function returns (stores) in `args` the command read by the program. E.g. command "`ls -l`" returns `args[0]="ls", args[1]="-l"` and `args[2]=NULL`. Variable `background` is set to 1 (true) if the command ends with '&'. The input arguments of this function are, an array of char and its size (`inputBuffer` y `size`). There is an example of how to use this function in `Shell_project.c`.

status, job_state, status_strings, state_strings

```
enum status { SUSPENDED, SIGNALED, EXITED};
enum job_state { FOREGROUND, BACKGROUND, STOPPED };

static char* status_strings[] = { "Suspended","Signaled","Exited" };
static char* state_strings[] = { "Foreground","Background","Stopped" };
```

These are two enum data-types to identify the termination cause and current state of a task. We have defined two string arrays to print them directly on screen. For example, a variable of type `status` "`state`" could report its value as follows:

```
printf("Reason for termination: %s\n", status_strings[state]);
```

analyze_status

```
enum status analyze_status(int status, int *info);
```

This function analyzes the integer `status` returned by function `waitpid()` and returns the cause of termination of a job (as an enumerated `status`) in addition to more information related to the termination (`info`).

job

```
typedef struct job_
{
	pid_t pgid; /* group id = process lider id */
	char * command; /* program name */
	enum job_state state;
	struct job_ *next; /* next job in the list */
} job;
```

The data type `job` is used to represent tasks and link them using a task-list. A task list is a pointer to `job` (`job *`) and a node in the list will also be handled as a pointer to `job (job *)`

Description of “**job_control**” module

new_job

```
job * new_job(pid_t pid, const char * command, enum job_state state);
```

This function creates a new task. The necessary input arguments to this function are: `pid`, `command` name and `status` (`BACKGROUND or FOREGROUND`).

new_list, list_size, empty_list, print_job_list

```
job * new_list(const char * name);
int list_size(job * list);
int empty_list(job * list);
void print_job_list(job * list);
```

These functions are used to create a new list (with a string as name) to determine the size of the list, to check whether the list is empty (returns 1 if empty) and print the contents of a list to screen. They are actually implemented as macros.

add_job, delete_job, get_item_bypid, get_item_bypos

```
void add_job (job * list, job * item);
int delete_job(job * list, job * item);
job * get_item_bypid  (job * list, pid_t pid);
job * get_item_bypos( job * list, int n);
```

These functions work on a task list. The first two add or delete a task list, `delete_job` returns 1 if it is successfully eliminated. The last two search for and return an element of the list by using its `pid` or position. They return `NULL` if not found. For example `add_job` can be used as follows:

```
job * my_job_list = new_list("Tareas Shell");
…
add_job( my_job_list, new_job(pid_fork, args[0], background) );
```

ignore_terminal_signals(), restore_terminal_signals()

```
void ignore_terminal_signals()
void restore_terminal_signals()
```

These functions (actually macros) are used to enable or disable signals related to the terminal, `SIGINT SIGQUIT SIGTSTP SIGTTIN SIGTTOU`. The Shell should ignore these signals, but the command created with `fork()` should restore its default behavior.

new_process_group

```
void new_process_group(pid_t pid)
```

This macro creates a new group of processes for the newly created process. The shell should carry it out after doing `fork`()

Description of "**`job_control`**" module
`set_terminal`
`void set_terminal(pid_t pid)` This macro assigns the terminal to a group of processes identified by pid. The terminal should be transferred to the foreground and recover for Shell when it expires or is suspended.
`block_SIGCHLD, unblock_SIGCHLD`

`void block_SIGCHLD()`
`void unblock_SIGCHLD()`
 	
These macros are useful to block signal "`SIGCHLD`" in the code-sections where the Shell modifies or accesses data-structures (the task list) that can be accessed through the handler of that signal and avoid accessing data in invalid or inconsistent states.

The following tasks are to be carried out by the student in this lab project. Each task must be delivered through Campus Virtual so that the instructor can evaluate them. The final task, which is the most complex one, will be presented and defended.

Task 1

The basis "Shell Project.c" is shown at the end of this document. The code enters a loop where it reads and analyzes command lines typed from the keyboard. The first stage focuses on the task of creating a process and making a new program run. You must add the necessary code to "Shell Project.c" in order to:

1. Execute the commands in an independent process
2. Wait or not to wait the termination of the command depending on whether the command is a foreground or background process
3. Print the information about the command-process, its `pid`, state, etc. to screen
4. continue the loop to process the next command

In this task the student should use the system calls `fork`, `execvp`, `waitpid` and `exit`. The information to be printed on the screen are: `pid`, name, type and code of termination or the signal of the termination reason. For example:

```
Foreground pid: 5615, command: ls, Exited, info: 0
Foreground pid: 5616, command: vi, Suspended, info: 20
```

For background commands should be reported their `pid`, name and indication that they are executing on background simultaneously with the Shell. For example:

```
Background job running... pid: 5622, command: sleep
```

When a command is not found and obviously cannot run, the Shell should print an appropriate error message. For example:

```
     Error, command not found: lss
```

To compile the program use either:

```
gcc –c Shell_project.c //This command will generate Shell_project.o
gcc –c job_control.c   //This command will generate job_control.o
gcc -o Shell Shell_project.o  job_control.o
```

or

gcc Shell_project.c job_control.c –o Shell

To execute the program use:

```
./Shell
```

To exit use: CONTROL + D (^D).

Task 2

A Shell should distinguish internal commands from external commands. Internal commands are implemented by the Shell itself and external commands are executable files that the Shell brings typically from `/bin` or `/usr/bin`. Up to this moment your Shell does not provide any internal command.

1. Now, we start implementing the `cd` internal command, which allows changing the current directory. To implement this command use function "`chdir`" .

2. To execute an external command in the Shell, the command should belong to an independent process group so that the terminal can be assigned to one unique foreground task at a time. Therefore, **the child processes of the shell are assigned a group id (**i.e., **gid)** that **differs from the parent id** using function `new_process_group()` function, wich is a macro that returns `setpgid()`.

3. A foreground **child process** must be **the unique owner of the terminal** since the parent process is blocked in `waitpid`. The terminal should be given back to the Shell using function `set_terminal()`, a macro that returns `tcsetpgrp()`. If the task is sent to the background, it should **not** be given the terminal, because the Shell should continue reading the keyboard, i.e. become the foreground job.

4. To make the Shell work correctly by giving the terminal to foreground tasks, it is necessary to ignore all signals related to the terminal. This can be done by including function `ignore_terminal_signals()` at the beginning of the program. When a new process is created, before execution, the signal handlers should be restored to the default operation (within the child process) using function `restore_terminal_signals()`.

5. The value returned by `waitpid` in `status` indicates whether the child process has terminated or is blocked. To detect the suspension of a child, the option WUNTRACED in function `waitpid` should be added. Make sure that the suspended foreground processes are reported by the Shell.

Task 3

Background commands terminate in a zombie state indicated by `<defunct>` when using the `ps` command. Up to now the Shell does not take care of them.

1. Please, install a handler for SIGCHLD to handle the termination / suspension of background tasks asynchronously using the `signal()` function. To do so, create a task list using the functions from the "`job_control`" module and add each command launched on background to that list.

2. When the handler of signal SIGCHLD is activated, we must review each list entry to see if any background process has terminated or is suspended. To check whether a process has completed without blocking the Shell, use the option WNOHANG in function `waitpid()`.

3. This exercise should avoid leaving background tasks in zombie state and make the Shell handle them and inform about their termination / suspension. When a task is terminated or suspended, the list of tasks should be updated adequately.

Task 4

The implementation should **safely manage background tasks** , i.e. launched with '&'. If the job is sent to the background it should **not be associated to the terminal**, because the Shell needs to continue reading the keyboard; the shell comes to the foreground.

Add a job control list using types and functions defined in module `job_control.c`. This list should allow controlling multiple running background tasks as well as multiple suspended tasks. The suspended tasks can come from foreground suspended tasks (ctrl. + Z) or background tasks that were suspended after trying to read from the terminal. Signal SIGSTOP will also suspend any foreground or background task.
The code we are designing is reentrant, i.e., while the Shell is running, signal SIGCHLD could arrive and thus the shell executes the handler.

1. To avoid consistency problems between child and parent when accessing the job-list, signal SIGCHLD should be blocked in the Shell-code wherever the job list is modified. Signal SIGCHLD can be blocked by functions (`block_SIGCHLD, unblock_SIGCHLD`) available in job_control.c .

2. The shell should **implement the following internal commands:**
   - **"jobs"** prints a list of background and suspended tasks, or a message if there is none. The job list should include both, as well the list of tasks explicitly sent to the background (with '&'), as the suspended (stopped) tasks. A function that prints a list of tasks already exists in `job_control`.

   - **"fg"** operates on the task list. **The command puts a suspended or background task to run on the foreground.** The argument of this command is the identifier of the place where that task in located in the task list (1, 2, ...). If the argument is not specified, it should be applied to the first task of the list (the last entry to the list). It is important that the signals are sent to the entire process group using `killpg()`, equivalent to function `kill` with a negative `pid;` please, study the help online.

```
Note: The operations to be performed on a background process,
once found in the list are: yielding the terminal to it (was
under parent control), change only the necessary things in its
job structure and send it a signal to continue, if it is
suspended, for example, waiting to get the terminal.
```

- **“bg”** works on the task list adding a task that is suspended running on the background. The argument of this command should be the identifier of the place of that process in the list (1, 2, 3, ...).

SOURCE CODE: Shell_project.c

```c
/**
UNIX Shell Project Operating Systems, Computer Architecture, UMA
Code adapted from "Operating System Concepts", Silberschatz et al.

To compile and run the program:
   $ gcc Shell_project.c job_control.c -o Shell
   $ ./Shell
	type ^D to exit program
**/

#include "job_control.h"   // remember to compile with module job_control.c

#define MAX_LINE 256 /* 256 chars per line, per command, should be enough. */

// -----------------------------------------------------------------------
//                            MAIN
// -----------------------------------------------------------------------

int main(void)
{
	char inputBuffer[MAX_LINE]; /* buffer to hold the entered command */
	int background;             /* equals 1 if a command is followed by '&' */
	char *args[MAX_LINE/2];     /* command line (of 256), max 128 arguments */
	// probably useful variables:
	int pid_fork, pid_wait; /* pid for created and waiting process */
	int status;             /* status returned by wait */
	enum status status_res; /* status processed by analyze_status() */
	int info;				/* info processed by analyze_status() */

	while (1)  /* Program terminates normally inside get_command() after ^D */
	{
	printf("COMMAND->");
	fflush(stdout);
	get_command(inputBuffer, MAX_LINE, args, &background);/*get next command*/

		if(args[0]==NULL) continue;   // if empty command

		/* the steps are:
		 (1) fork a child process using fork()
		 (2) the child process will invoke execvp()
		 (3) if background == 0, the parent will wait, otherwise continue
		 (4) Shell shows a status message for processed command
		 (5) loop returns to get_command() function
		*/
	} // end while
}
```

Appendix: **Concepts of job control in UNIX**

The main goal of an interactive shell is to read commands from the terminal typed by the user and to create processes executing the programs specified by these commands. As mentioned before, the shell can do so by using `fork` and `exec` system-calls.

When a command is launched, all processes and their children performing the task, form a *process group*. For example, a simple compilation command such as `'cc -c shFSO.c'` may launch multiple processes (*preprocessing, compilation, assembly and linking*) to complete the compilation task.

All the processes belong to a process group. When a process is created using fork, it will be a member of the same process group as its parent process. You can put a new process in another process group using system call `setpgid`.

In UNIX-like Operating systems, processes are organized into **session** sets. The **session** ID is the same as the `pid` of the process that created the session through system call `setsid()`. That process is known as the **session leader** of the session group. All children of that process are members of the session unless they specifically remove themselves from it. Function `setsid()` does not have any arguments and returns the new session ID.

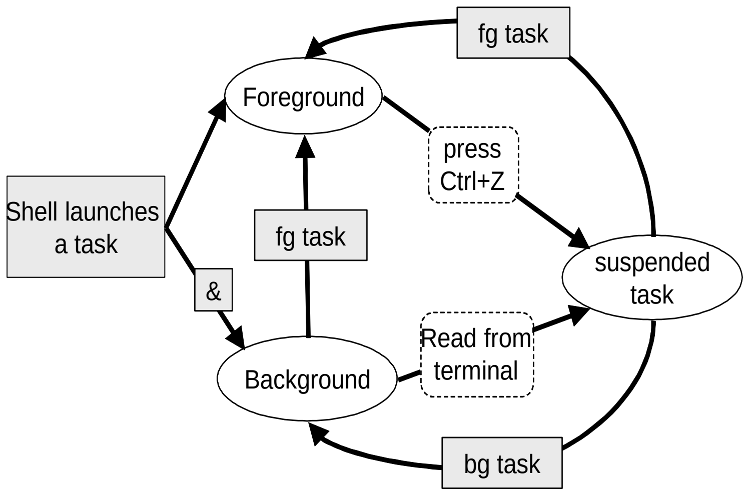
A new session is created when we login to the system and the shell becomes the first process (leader) of the session. All process groups created by the shell become members of its session. Every session is tied to a terminal from which processes in the session get their input and to which they send their output. That terminal may be the machine local console, a terminal connected over a serial line etc. The terminal to which a session is related is called the **controlling terminal** (or controlling tty) of the session. A terminal can be the controlling terminal for only one session at a time.

A UNIX shell with job control must manage and select which task the terminal can use at each moment to perform its input/output. If it were not so, multiple tasks may try to read at the same time from the terminal and an uncertainty would arise about which task should eventually receive the input typed by the user at the terminal. To prevent such confusion, the shell must ensure an appropriate use of the terminal using function `tcsetpgrp`.

The shell will give access and an unlimited use of the terminal to one single group of processes (tasks) at a time. The process group that is using the terminal is known as a **foreground task** (*foreground job*). The process groups that have been launched by the shell and running without access to the terminal are called **background tasks**. Therefore, in one session there can only be one active foreground task. The shell will be the foreground task while it accepts user commands. The Shell itself is responsible for transferring the control of the terminal to each task that need to execute on foreground. When a foreground task terminates or is suspended, the shell regains the control of the terminal and newly becomes a foreground task that reads the next user command.

The shell not only has to control which tasks are running on foreground and which tasks are running on background, it also has to control when a task is **suspended**. If a background task tries to read/write from/to the terminal (to which it does not have access since it is a background process), the terminal driver suspends the task (if the operating mode `TOSTOP` is active terminal). In addition the user can suspend the foreground task by pressing (CTRL + Z) on the keyboard. A program can suspend any task by sending signal SIGSTOP to that task. It is the responsibility of the shell to notify when tasks are suspended and provide mechanisms to allow the user to interactively continue the execution of suspended jobs and decide whether a task should continue on the foreground or background (using the commands: `fg`, `bg` and `jobs`).

Here is an outline of how the task control of our shell works:



It is the responsibility of our shell to place a task in the foreground or background when launching it, by assigning it or not the control of the terminal. When a task is suspended by the terminal (by pressing ctrl. + Z or when it attempts to read/write from/in the terminal on the background) the shell has to detect this event (by analyzing the `status` returned by `wait)` and notify the suspension of the task by means of a message. The shell should implement internal commands:

- `Jobs`: lists background tasks by indicating whether they are running or suspended. Suspended task could also come from foreground and must be listed.
- `Fg`: continues running a foreground task (the task was suspended or running on the background)
- `Bg`: continues running a background task (the task was suspended)

Finally, notice that all the processes grouped into a task (process group), can be controlled by a group of signals. For example when you press CTRL + C the terminal's driver sends signal SIGINT to all processes that are part of the foreground task to finalize. Similarly other task control signals also send the signal to all the set of processes that form a task (process group). Some of these signals are:

`SIGINT`	INTR (CTRL+C) *interrupt* from the terminal
`SIGQUIT`	QUIT (CTRL+\) *quit* from the terminal
`SIGTSTP`	SUSP (CTRL+Z) *stops* from the terminal
`SIGTTIN`	background task tries to read on terminal
`SIGTTOU`	background task tries to write on terminal