**OPERATING SYSTEMS**

**The Shell and twc command**

**Name:** John Aldo Cuesta Agila

**Course section:** 2

**Deadline:** Tuesday, December 3, 2019

**Anexo: Detalles de Proyecto**

**Project Goals**

* To familiarize yourself with the Linux programming environment.
* To develop programming skills in C.
* To understand the relationship between OS command interpreters (shells), system calls, and the kernel.
* To design and implement an extremely simple shell.
* To learn how processes are handled (i.e., starting and waiting for their termination).
* To design and implement multithreaded programming strategies.

**Background**

The OS **command interpreter** is the program that people interact with in order to launch and control programs. On UNIX systems, the command interpreter is usually called the shell: it is a user-level program that gives people a command-line interface to launching, suspending, and killing other programs. sh, ksh, csh, tcsh, bash, ... are all examples of UNIX shells. (It might be useful to look at the manual pages of these shells, for example, type: "man csh".)

Every shell is structured as the following loop:

1. print out a prompt.
2. read a line of input from the user.
3. parse the line into the program name, and an array of parameters.
4. use the **fork()** system call to spawn a new child process.

* the child process then uses the exec() system call to launch the specified program.
* the parent process (the shell) uses the **wait()** system call to wait for the child to terminate.

1. once the child (i.e. the launched program) finishes, the shell repeats the loop by jumping to 1.

Although most of the commands people type on the prompt are the name of other UNIX programs (such as ls or more), shells recognize some special commands (called internal commands) which are not program names. For example, the exit command terminates the shell, and the cd command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking a child process to handle them.

**The Assignment**

This assignment consists of two parts. In the first, you will design and implement an extremely simple shell that knows how to launch new programs, and also recognizes two internal commands (exit and cd), which we will describe below. The two internal commands will work by calling existing system calls (exit and chdir); in the second part of this assignment, you will develop an improved version of the wc command (you work in previous assignments) using threads, the wc command (work count) gets the word and line count.

**Part 1: Build a new shell**

Write a shell program in C which has the following features:

* It should recognize two internal commands, exit and cd. exit terminates the shell, i.e., the shell calls the exit() system call or returns from main. cd uses the chdir system call to change to the new directory.
* If the command line is not an exit or cd, it should assume the it is of the form

<executable name> <arg0> <arg1> .... <argN>

Your shell invokes the executable, passing it the command line

* Assume that the full path names, like /bin/ls, are given. Also, try to use the same prompt as in the following:

**SO<student\_id>sh**% /bin/date

Mon 11 Nov 2019 15:54:24 -05

**SO<student\_id>sh**% /bin/cat /etc/hostname

docente

**Note:** The words in bold are output by the shell (you should replace <student\_id> with your student id number), and the words underlined are typed in by the user.

* To allow users to pass arguments to programs you will have to parse the input line into words separated by whitespace (spaces and '\t' tab characters) and place these words into an array of strings. You might try using strtok() for this (man strtok for a very good example of how to solve exactly this problem with strtok).

Then you will need to pass the name of the command as well as the entire list of tokenized strings to one of the other variants of exec, such as execvp(). These tokenized strings will then end up as the argv[] argument to the main() function of the new program executed by the child process. Try man execv or man execvp for more details.

Please take a look at the manual pages of execv, fork and wait.

**Part 2: An improved version of the wc command: twc**

Design and develop an improved version of the wc command. The wc command (work count) gets the word and line count for each file. Your improved version of wc will be called iwc and it receives the below arguments:

twc [-lm] [file ...]

-w The number of words in each input file is written to the standard output.

-l The number of lines in each input file is written to the standard output.

Note: A word is a non-zero-length sequence of characters delimited by white space.

You will write a program in C that uses the pthread library to maximize the use of the available cores in the system. Hence, you program must query the available resources in the system. You will design and implement a strategy to split the data and the processing task. To each data-chunk, you will assign a thread, so that, it counts the requested information. The total count will be kept in a shared variable among the threads. Be aware of any race conditions while designing your strategy.

**Recommendations**

**Defensive programming** is an important concept in operating systems: an OS cannot simply fail when it encounters an error. It must check all parameters before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner. By “reasonable”, this means that you should print a meaningful and understandable error message and either continue processing or exit, depending upon the situation.

**Grading**

This assignment must be submitted via SidWeb with the following elements:

* Your code should be well documented in terms of comments. For example, good comments in general consist of a header (with your name, course section, date, and brief description), comments for each variable, and commented blocks of code.
* A Makefile for compiling your source code, including a clean directive.
* A diagram(s) that explains how your strategy works.
* A README file with some basic documentation about your code. This file should contain the following three components:
* Design Overview: A few paragraphs describing the overall structure of your code and any important structures.
* Complete Specification: Describe how you handled any ambiguities in the specification. For example, for this project, explain how your shell will handle lines that have no commands, double spaces between arguments, tabs, etc.
* Known bugs or problems: A list of any features that you did not implement or that you know are not working correctly.
* A brief (less than a page) discussion of any important design decisions that you made while implementing your shell and/or twc command.

Submit a compressed file (\*.zip) to the SidWeb and name it as follows:

OS.P1.2019\_2.<Lastname><Firstname>.zip

For example: **OS.P1.2019\_2.LopezAngel.zip**

The grade on the project will be calculated as follows:

* Shell: 30 points
* Command twc: 40 points
* Documentation: 10 points
* Strategy: 20 points

**Informe de proyecto**

**Nombre:** John Aldo Cuesta Agila

**Paralelo:** 2

**Fecha de entrega:** 3 de diciembre del 2019

**Parte 1: Construir un nuevo interpretador de comandos**

**Resumen del problema**

Se plantea desarrollar una consola de comandos que pueda manejar dos comandos: exit y cd.

El comportamiento de ambos comandos debe ser similar al que tienen al ser ejecutados en una consola común de un sistema operativo cualquiera.

Adicionalmente, el comando cd debe mostrar un formato específico que está especificado en el Anexo de Detalles del documento del proyecto. Además, se debe cumplir con la estabilidad de la ejecución de los procesos y el manejo de errores.

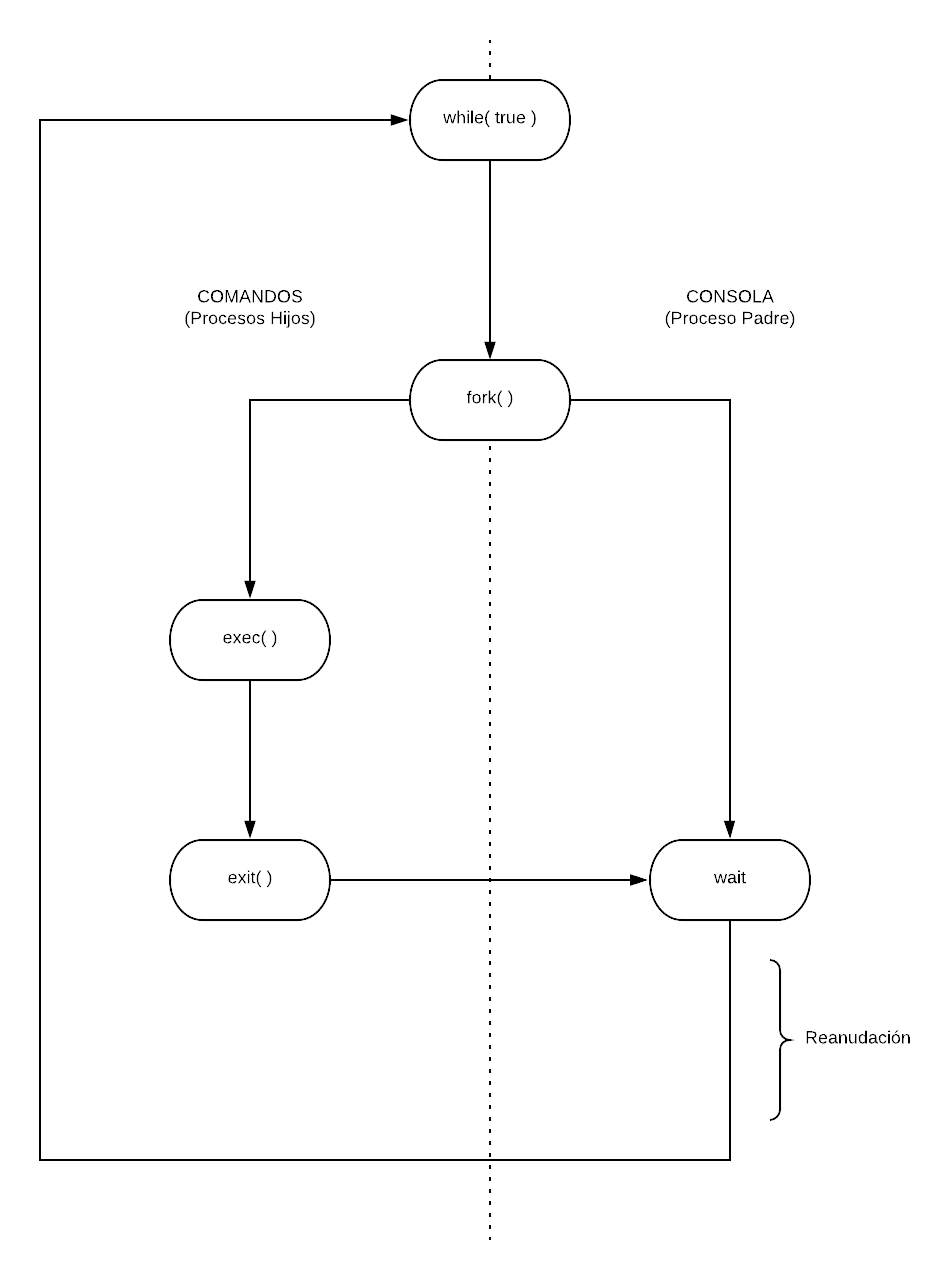
**Estrategia**

El planteamiento de la solución para esta parte del proyecto es el de mantener activo un proceso padre, es decir, la consola como tal. Además, se deben ejecutar procesos hijos, o los comandos de la consola, que deben ejecutar sus propias tareas.

Mientras los procesos hijos se crean y se ejecutan, el proceso padre debe mantenerse vigente. Cuando finalmente el proceso hijo termina, el proceso padre espera para mostrar el resultado del proceso hijo.

Finalmente, cuando el proceso padre termina todo lo que se está ejecutando se cierra en ese momento.

**Diagrama**



**Parte 2: Versión mejorada del comando wc: twc**

**Resumen del problema**

Se necesita desarrollar una versión mejorada del comando wc que pueda contar palabras y líneas en un archivo. El programa debe escribirse en C y utilizar la biblioteca pthread para maximizar el uso de los núcleos disponibles en el sistema. La versión mejorada debe llamarse twc y debe recibir los siguientes parámetros para ejecutar diferentes acciones:

**-w:** contar el número de palabras en un archivo

**-l:** contar el número de líneas en un archivo

El formato de la entrada en consola debe ser la siguiente:

twc [-lm] [file ...]

**Estrategia**

Para la solución se decide a particionar el archivo en chunks, estos serán porciones del archivo que dependen de la cantidad de hilos que se utilicen. Por ejemplo, si se tienen 4 hilos se tendrán 4 chunks. La cantidad de hilos se definen en “pthread\_create”.

Para realizar el conteo de líneas se busca el carácter “\n” o salto de línea, dependiendo de la cantidad de estos caracteres se obtiene la cantidad de líneas.

Para realizar el conteo de palabras se utilizan punteros para saber si el carácter actual es diferente de espacio y si el carácter siguiente es un espacio. Si estas condiciones se dan, entonces se considera que es una palabra. Para verificar si hay una palabra en los límites de los chunks se utiliza la misma estrategia. Este algoritmo sigue la base del “wc” genérico con algunas modificaciones.

Dado que no hay operaciones de escritura, y se utilizan diferentes punteros para leer el archivo, ya que para cada hilo se obtiene una dirección de memoria diferente, por ende cada puntero es independiente uno del otro y no se manejan zonas críticas, por lo que no hubo la necesidad de utilizar la exclusión mutua.

Esto se debe a que se realiza una lectura y no realiza conflictos o condiciones de carrera. Cuando se manejan multihilos se divide la longitud del archivo para la cantidad de hilos totales. El último chunk es el que va a tener más de la participación o menos de la participación.

Esto depende de la división ya que la división en C es de paquetes enteros. Cuando se tenga el tamaño del bloque, cada hilo se inicializa en estado “joineable” para poder esperar los hilos que van contando y cuando terminen se puedan juntar con los demás hilos.

Cuando se tengan los chunks contados al final, se cuentan las partes que se pudieron saltar. Con la validación se maneja los caracteres de inicio y fin para controlar si lo que se tiene es una palabra. De manera global sólo se acumulan las particiones para dar el resultado.

Este algoritmo es eficiente ya que no carga en memoria más de lo necesario. Esto se refleja especialmente en archivos muy grandes. Al contrario de utilizar, por ejemplo, mmap para separar espacio en la RAM entonces se está utilizando más memoria para el archivo de gran contenido.

**Diagrama**

