Universidad de San Andrés

I301 Arquitectura de computadoras y Sistemas Operativos

TP-4 Procesos

Entrega: **** 11:59 PM

Importante: La resolución del TP es de manera individual.

Comunicación básica entre procesos

Como ***warm up*** para este primer ejercicio, el objetivo es implementar un esquema de comunicación en forma de anillo para interconectar los procesos. En un esquema de anillo se da con al menos tres procesos están conectados formando un bucle cerrado. Cada proceso está comunicado exactamente con dos procesos: su predecesor y su sucesor. Recibe un mensaje del predecesor y lo envía al sucesor. En este caso, la comunicación se llevará a cabo a través de ***pipes***, las cuales deben ser implementadas.

Al inicio, alguno de los procesos del anillo recibirá del padre un ***número entero*** como mensaje a transmitir. Este mensaje será enviado al siguiente proceso en el anillo, quien, tras recibirlo, lo incrementará en uno y luego lo enviará al siguiente proceso en el anillo. Este proceso continuará hasta que el proceso que inició la comunicación reciba, del último proceso, el resultado del mensaje inicialmente enviado.

Se sugiere que el programa inicial cree un conjunto de procesos hijos, que deben ser organizados para formar un anillo. Por ejemplo, el hijo 1 recibe, del padre, el mensaje, lo incrementa y lo envía al hijo 2. Este último lo incrementa nuevamente y lo pasa al hijo 3, y así sucesivamente, hasta llegar al último hijo, que incrementa el valor por última vez y lo envía de vuelta al proceso padre. Este último debe mostrar el resultado final del proceso de comunicación en la salida estándar.

Se espera que el programa pueda ejecutarse como:

./anillo <n><c><s>,

donde:

<n> es la cantidad de procesos hijos del anillo.
<c> es el valor del mensaje inicial.
<s> es el número de proceso que inicia la comunicación

Les proveemos un archivo **ring.c** deberían trabajar sobre ese archivo pero sientanse libres de modificarlo e incluso hacer sus propias librerías.

IMPORTANTE: para que les sirva como referencia la solución del problema programada por nosotros tiene del orden de 40 líneas de código.

SHELL

1. Introducción

En este práctico, se busca el desarrollo de un shell interactivo en lenguaje C. Un shell, en esencia, es un programa que ejecuta otros programas en respuesta a comandos introducidos mediante texto. Dado que el objetivo principal es evaluar el conocimiento sobre aspectos fundamentales de sistemas operativos, se ha optado por omitir la tarea de procesar cadenas de texto. El parsing es crucial en un shell, ya que un algoritmo sólido para esta tarea simplifica considerablemente el diseño de la interacción entre el software y el sistema operativo.

2. Syscalls

Como seguramente ya estarán intuyendo, un shell es un software que actúa como interfaz con el sistema operativo, facilitando la interacción de los programas a nivel usuario con el SO. Esta interacción se lleva a cabo mediante *syscalls* (llamadas al sistema), es decir, cuando un programa requiere realizar una tarea que demanda privilegios de nivel de sistema operativo, como leer o escribir en el disco, enviar datos a través de la red o crear un nuevo proceso, no puede acceder directamente a los recursos del sistema. En su lugar, realiza una llamada al sistema, que consiste en una solicitud al *kernel* para que realice la operación en su nombre. La Figura 1 intenta ilustrar esta idea (pueden encontrar más información al respecto aquí[1]):

[1] https://en.wikipedia.org/wiki/POSIX_terminal_interface

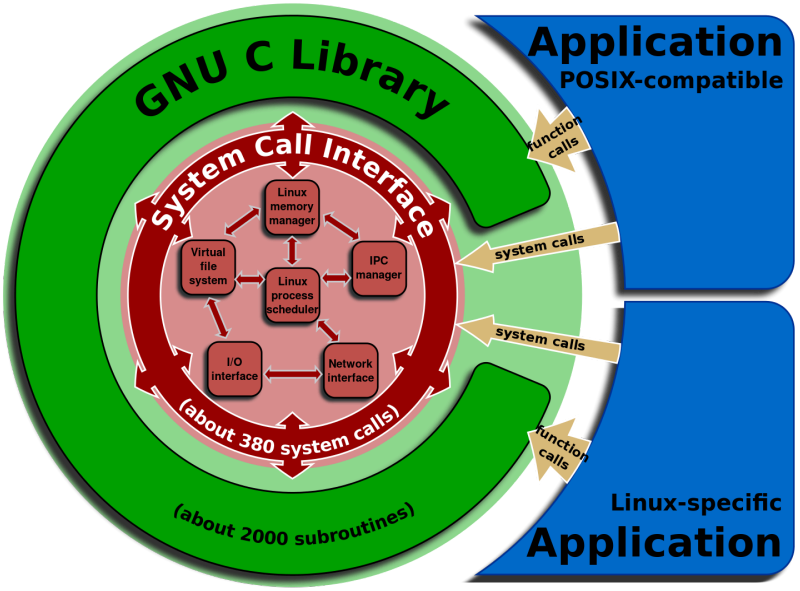

Fig 1. La biblioteca GNU C es un *wrapper* alrededor de la interfaz del kernel de Unix.

La biblioteca **GNU C** encapsula estas llamadas al sistema mediante funciones que siguen el estándar ***POSIX***[2] que ya estuvieron utilizando, por ejemplo: **fork, open, read, write, dup, dup2, waitpid, execvp, close, etc.** Recuerden que ejecutando en su terminal el comando:

man 2 execvp

pueden obtener información sumamente útil y detallada sobre la funcionalidad de los diferentes syscalls. Otro punto importante, es que lean la documentación en el manual de linux de **TODAS** las syscalls que van a estar utilizando, sobre todo hagan hincapié en entender que hace **waitpid** y **close** para usarlas de manera correcta porque les va a ahorrar bastantes dolores de cabeza.

3. Objetivos y funcionalidad

En concreto, el código que deben programar debe resolver el siguiente problema: dada una cadena de programas separados mediante pipes “|”, su programa debe poder reproducir la funcionalidad de su bash.

[2] https://en.wikipedia.org/wiki/POSIX

Ejemplos:

a. El comando:

ls | grep .zip

en Linux realiza lo siguiente:

- **ls**: Lista los archivos y directorios en el directorio actual.
- **|:** Este símbolo, llamado *pipe*, redirige la salida del comando ls hacia la entrada del siguiente comando.
- **grep ".zip"**: Filtra la entrada recibida, mostrando solo las líneas que contienen el texto **".zip"**.

Les ***recomendamos (muy fuertemente)*** que prueben con más comandos de este estilo y asegurense de que entienden lo está ocurriendo, ya que el software que nos van a entregar debe ser capaz de reproducir la misma salida que genera su propio bash. Si lo desean, a modo de consulta, pueden utilizar este recurso Pure Bash Bible. Nosotros vamos a evaluar su TP generando comandos con este estilo que y usando más programas unidos por más pipes.

Su TP debe ser capaz de resolver comandos de este tipo (nótese que NO tiene comillas):

ls | grep .zip

Consideraremos como ***extra-credit*** si resuelven este tipo de comandos:

ls | grep “.zip”

ls | grep “.png .zip”

(Nótese que cuando se introducen espacios para buscar un patrón hacen falta sí o sí las comillas).

4. Comentarios

Dentro del *bundle* de archivos que van a recibir hay dos particularmente importantes:

Makefile: Ejecutando **make** pueden compilar el **.c** recuerden que si agregan archivos deben modificarlo o crear otro Makefile.

shell.c: contiene **main** que realiza la ejecución del shell, sobre este archivo deben trabajar, pueden modificarlo a como quieran o crear otros archivos.

Para que les sirva como referencia la solución del problema programada por nosotros tiene del orden de 80 líneas de código.

Un punto importante es notar que **execvp()** espera como primer argumento el nombre del programa que se desea ejecutar y como segundo argumento un array de punteros a caracteres que contiene los argumentos del comando. Hagan el ejercicio de entender la información que les provee el manual de la linux.

Ejemplo:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
   // Array de punteros a caracteres que contiene los argumentos del
comando
   char *args[] = {"ls", "-l", NULL};

   // Ejecutar el comando "ls -l"
   execvp(args[0], args);

   // Si execvp retorna, significa que ocurrió un error
   perror("Error executing command");
   exit(EXIT_FAILURE);
}
```

Importante:

La entrega se hará a través de un repositorio de github suyo que deberá contener todo el código necesario para que nosotros podamos compilar y correr el proyecto. Deberían hacernos colaboradores de ese repositorio, al menos, hasta que le demos la corrección.