Universidad Nacional de General Sarmiento

Sistemas Operativos y Redes I
Comisión 2 Mañana

Trabajo Práctico

Sistemas Operativos y Redes I
TP1 1er Cuatrimestre 2019

Andrés Rojas Paredes

Hvara Ocar

APELLIDO Y NOMBRE	LEGAJO	EMAIL
Tula, Ignacio Mariano	35.226.620/2014	itula@logos.net.ar itula@ungs.edu.ar

1era Parte
Shell

Ejercicio 1. Sistema Operativo GNU/Linux

Inciso 1

Se instaló Ubuntu 18.10 en la notebook personal del estudiante, y una versión en un pendrive USB con persistencia.

El único inconveniente detectado es que se prefirió Ubuntu por sobre Debian, puesto que en esta última distribución existían problemas con los controladores de los adaptadores de red y de gráficos.

En ambas instalaciones, se ejecutó los comandos que quedaron descritos en un archivo bash ubicado en ***/home/sor_user/tp1/proyectosot1ungstula/ejercicio1.sh***

```
#!/bin/bash
sudo adduser sor_user sudo                          # inciso 2
su sor_user                                         # inciso 3
sudo apt-get update                                 # inciso 3
sudo apt-get install gcc vim nano gedit emacs git # inciso 4
```

Ejercicio 2. Shell y Supermenu

Inciso 1
Se creó el archivo ***supermenu.sh*** en ***/home/sor_user/tp1/proyectosot1ungstula .***

Inciso 2
Se creó en Gitlab el proyecto ubicado en la ruta:
https://gitlab.com/itula/proyectosor1ungstula

El cual se clonó en la ubicación ***/home/sor_user/tp1/*** mediante el comando:

```
git clone https://gitlab.com/itula/proyectosor1ungstula.git
```

Se modificó la variable del archivo ***supermenu.sh***

```
proyectoActual="/home/sor_user/tp1/proyectosor1ungstula"
```

Es decir que se incluyó el propio supermenu dentro del proyecto de Gitlab.

Fue necesario ejecutar en consola también:

```
git config --global user.email "itula@logos.net.ar"
git config --global user.name "Ignacio Tula"
```

Inciso 3
Para agregar una opción nueva en el ***supermenu.sh*** es necesario realizar tres modificaciones en el código fuente. A saber:

1. En la definición de la función "*imprimir_menu*" se debe agregar una cadena de caracteres que es el texto que leerá el usuario para comprender qué letra debe presionar para realizar determinada tarea.

```
35. echo -e "\t\t\t p.  Mostrar PCB de proceso";
```

2. Agregar una posibilidad dentro del *case* del bloque de la "*lógica principal"* que detecte el carácter asignado a la determinada tarea para que ejecute una función.

```
181. p|P) p_funcion;;
```

3. Agregar la definición de la función que es llamada en el case del item 2.

```
p_funcion () {
    imprimir_encabezado "\tOpción p. Mostrar PCB";
            echo -e "Vamos a ejecutar TOP, para que elija el PID que
quiere analizar."
            decidir "top";
            imprimir_encabezado "\tOpción p. Mostrar PCB";
            echo -e "Indique el PID del cual desea conocer su PCB";
            read pcb;
            post_p_funcion $pcb;
   }

post_p_funcion() {
    imprimir_encabezado "\tOpción p. Mostrar PCB";
     decidir "cat /proc/$1/status | grep 'Name\|Pid\|Ppid\|State\|
```

```
voluntary_ctxt_swi tches\|nonvoluntary_ctxt_switches'";
}
```

En este ejercicio en particular se decidió que se ejecuten en dos funciones. Una primera que permitiera ver todos los procesos con *TOP* y una segunda que al ser finalizado *TOP* preguntara por el *PID* que se deseaba conocer.

2da Parte

Procesos y Semáforos

Ejercicio 1. Procesos y Fork

Inciso 1

```
1. #include <stdio.h>
2. #include <sys/types.h>
3. #include <unistd.h>
4.
5. int main() {
6.   fork();
7.
8.  printf("Hola Mundo!\n");
9.
10.  return 0;
11. }
```

Este programa emitirá dos líneas de texto en la consola expresando “Hola Mundo!". Esto es debido a que un printf es ejecutado por el proceso principal, y otro por el proceso hijo que es llamado a través de fork(). Al no existir una estructura de control, ni una variable sobre la cual reconocer el pid del proceso actual, ambos procesos (padre e hijo) ejecutarán lo mismo, desde la línea 7 en adelante.

Inciso 2. proyectosor1ungstula/enunciado2/ejercicio1/bombafork.c

```c
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int numero = 0;

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


while (1) {

  fork();

  printf("Ahora el numero es:%d \n", numero);
  numero+=1;
}

      return 0;
}
```

En las ejecuciones realizadas no se ha encontrado límite alguno para la creación de procesos, o al menos no se ha tocado el límite que pueda tener el Sistema Operativo. También su medición es complicada, considerando que la computadora deja de ser capaz de mostrar otras aplicaciones de monitoreo del procesador y memoria, hasta el punto de tildarse por falta de memoria RAM principal disponible.

Inciso 3. proyectosor1ungstula/enunciado2/ejercicio1/tush.c

Se creó “Tush” el intérprete de comandos shell de Tula.

Ejercicio 2. Threads y semáforos

Inciso 1. proyectosor1ungstula/enunciado2/ejercicio2/escritorLector.c

Se implementó un programa que posee 2 hilos, uno para escritura y otro para lectura. Comparten dos variables. Una llamada numeroActual, que sirve para almacenar un número que se obtiene por medio del azar. Y otra variable adicional que guarda el número anterior seleccionado por el azar.

Inciso 2

Ambos hilos, por una cuestión para facilitar la interpretación de este trabajo muestran un texto en pantalla, indicando si el mismo es el escritor o el lector, y luego las variables compartidas.

La condición de carrera entre los hilos existe en el momento en el cual el escritor se encuentra en el proceso de modificar ambas variables. Si el escritor al reemplazar la variable del número anterior por el actual, es interrumpido allí por el planificador del sistema operativo, y se ejecuta el hilo lector este mismo leerá el mismo valor en ambas variables.

Suponiendo una secuencia de números: 1 -> 3 -> 8 -> 5

Se esperaría que un proceso lector muestre (suponiendo que se ejecuta siempre que un escritor haya finalizado).

```
Soy el lector | El actual es: 1. El anterior es: 0
Soy el lector | El actual es: 3. El anterior es: 1
Soy el lector | El actual es: 8. El anterior es: 3
Soy el lector | El actual es: 5. El anterior es: 8
```

Pero si no se puede controlar la secuencia de ejecución, pueden ocurrir anomalías como estas:

```
Soy el lector | El actual es: 1. El anterior es: 0
Soy el lector | El actual es: 1. El anterior es: 1
Soy el lector | El actual es: 3. El anterior es: 1
Soy el lector | El actual es: 5. El anterior es: 8
```

De forma igual pero inversa, el lector puede ser interrumpido por el planificador mientras obtuvo de memoria el valor actual, pero aún sin conseguir el valor anterior. Si durante esta interrupción, un escritor se ejecutase completamente, pueden ocurrir nuevas situaciones no deseadas como la anterior.

Inciso 3

```
void* escritorDeNumero() {

    while(1) {


        // INICIO DE LA SECCIÓN CRÍTICA
        numeroCompartidoAnterior = numeroCompartido;
        usleep(500);
        numeroCompartido = rand() % 1000;
     printf("\nsoy el escritor    (%d,%d)",
           numeroCompartido, numeroCompartidoAnterior);
        // FIN DE LA SECCIÓN CRÍTICA

    }

}
```

```
void* lectorDeNumero() {

    while(1) {

        // INICIO DE LA SECCIÓN CRÍTICA
        printf("\nsoy el lector | El actual es: %d. El anterior es:%d",
numeroCompartido, numeroCompartidoAnterior);
       // FIN DE LA SECCIÓN CRÍTICA

    }

}
```

Inciso 4. proyectosor1ungstula/enunciado2/ejercicio2/escritorLectorMutex.c

Una forma sencilla de resolver el problema con mutex es:

```
pthread_mutex_t excluMutua; // Para controlar la lectura mientras haya
modificación, o impedir la modificación mientras hay lectura
```

```
void* escritorDeNumero() {

    while(1) {

        //Escritura del nuevo numeroCompartido y resguardo del anterior.

        pthread_mutex_lock(&excluMutua);
        numeroCompartidoAnterior = numeroCompartido;

        numeroCompartido = rand() % 1000;
        printf("\nsoy el escritor   (%d,%d)",
               numeroCompartido, numeroCompartidoAnterior);
        pthread_mutex_unlock(&excluMutua);
        usleep(25000);
    }

}

void* lectorDeNumero() {


    while(1) {



        //Lectura del numeroCompartido y mostrar en pantalla
        pthread_mutex_lock(&excluMutua);

        printf("\nsoy el lector | El actual es: %d. El anterior es:%d\n",
              numeroCompartido, numeroCompartidoAnterior);

        pthread_mutex_unlock(&excluMutua);
        usleep(10000);
    }

}
```

Ahora bien, para una ejecución simultánea de un único hilo lector y un único hilo escritor se ha encontrado que esta solución funciona, pero si sería necesario agregar semáforos de sincronización para poder ejecutar m hilos de lectura y n hilos de escritura.

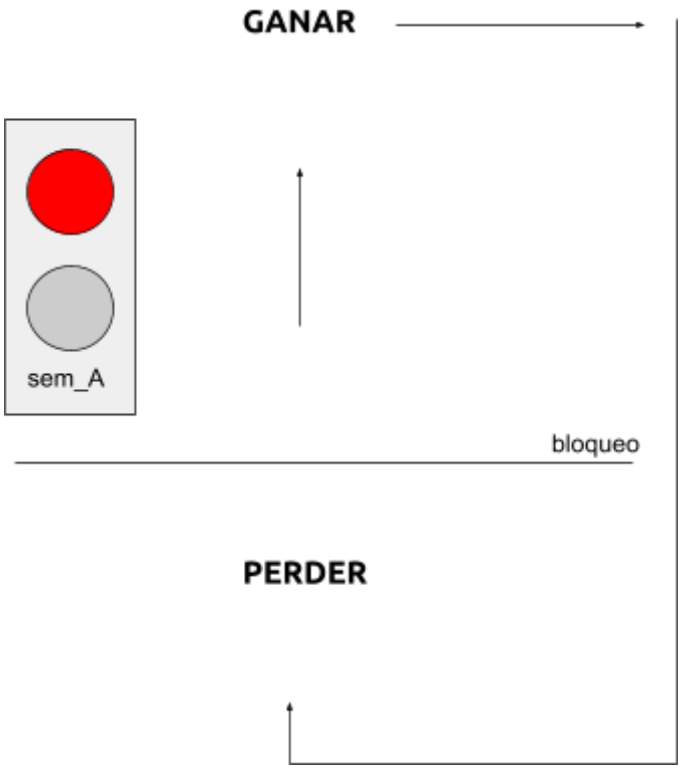
Se implementa esta solución en el archivo
proyectosor1ungstula/enunciado2/ejercicio2/escritorLectorSemaforos.c

Inciso 5. proyectosor1ungstula/enunciado2/ejercicio2/juegoDeLaVida.c

Inciso 6. proyectosor1ungstula/enunciado2/ejercicio2/outputJuegoDeLaVida.txt

Se guardó el output de la ejecución con un parámetro de 500. No existe el azar, los resultados “ganar” y “perder” se van alternando en una secuencia que comienza por el primer hilo en ser ejecutado.

Esto ocurre porque una vez que “ganar” encontró habilitado su semáforo primero (antes que “perder”), para cuando vuelve a comenzar y esperar por el mismo semáforo, quien está ahora en la cola primero es perder.



Inciso 7.

No existen rachas ganadoras o perdedoras, la justicia que imparte el planificador de procesos es tal que permite que sus ejecuciones sean alternadas, a tal punto que apenas

queda habilitado el semáforo *sem_A,* es entregado a quien continúa primero en la lista de pendiente o de hilos bloqueados por el semáforo.

Inciso 8. **proyectosor1ungstula/enunciado2/ejercicio2/juegoDeLaVidaBienJugado.c**
El agregar prioridad al hilo ganador, modifica levemente el resultado, pero solo luego de muchas repeticiones (se utilizó 100.000) y no siempre la diferencia fue a favor del hilo ganador.

Pareciera que la prioridad modifica el tiempo que se asigna a la ráfaga de ejecución pero la lógica de los semáforos es invulnerable a esta prioridad.

Ejercicio 3. El poder del paralelismo

Inciso 1.

proyectosor1ungstula/enunciado2/ejercicio3/

La implementación con paralelismo de datos es : *paralelismoDatos.c*
La versión secuencial de la misma es: *paralelismoDeDatosNOsecuencial.c*

La implementación con paralelismo de tareas es : *paralelismoTarea.c*
La versión secuencial de la misma es: *paralelismoDeTareaNOsecuencia.c*

La versión con paralelismo de tareas tiene un promedio de 0,015s de ejecución. Con picos máximos de 0,023s y mínimos de 0,005s

La versión secuencial de paralelismo de tareas tiene un promedio de 0,005s

:-(