En este trabajo, se realizaron varias actividades relacionadas con los temas vistos en la materia con el fin de reforzar los conocimientos adquiridos.

En primer lugar, se trabajó con el shell, y un supermenu, en el cual se fueron añadiendo funciones que facilitan la ejecución de diversas tareas, como por ejemplo: crear o borrar 100 carpetas dentro del sistema. Además, el supermenu contaba con la función de sincronizarse con Gitlab a través de un proyecto, en el cual se permite ver el estado del proyecto, y además subir archivos en el mismo.

Por otro lado, a través de un servidor Debian creado dentro de la universidad, con el software “Putty” se accedió al mismo con el fin de testear el supermenu y facilitar el uso del sistema operativo.

En segunda instancia, se creó un programa codificado en lenguaje C, el cual realiza operaciones aritméticas (suma), y operaciones de I/O (leer el input del usuario). Luego de hacer esto, se usó la herramienta *htop*, con la cual se pueden ver los estados por los que pasa el programa.

En la siguiente captura de pantalla, se observa al proceso esperando un valor de entrada:



En esta otra imagen se observa al proceso en ejecución:



Por otro lado, se utilizó la herramienta web *“Trello”*, la cual permite organizar las tareas por realizar de forma sencilla y ordenada, clasificando las tareas según el estado en el que se encuentren (nuevo, listo para hacer, running, esperando y terminado).



En tercer término, se presentó un programa ya armado, en el cuál se pedía que se interpretara el output que devolvía el mismo, además de calcular cuantas veces se repetiría el String *“¡Hola mundo!”* para n veces. El análisis realizado es el siguiente:

* Con i = 0, se creará un proceso A, y se imprimirá dos veces *“¡Hola mundo!”*, ya que el comando lo realizará el proceso padre y el A.
* Con i = 1, el proceso padre y el proceso A tendrán un nuevo proceso hijo cada uno, y luego los cuatro procesos imprimirán *“¡Hola mundo!”*.
* Con i = 2, la cual es la última iteración, todos los procesos tendrán un hijo cada uno, quedando así ya ocho procesos en total, y cada uno imprimirá *“¡Hola mundo!”*.

Como conclusión de este seguimiento, sacamos que para n iteraciones se crearán 2^n procesos, y se imprimirá “2^n + 2^n – 1 + ... + 2^2 + 2^1” veces *“¡Hola mundo!”*.

El código anterior podría convertirse en una bomba fork en si el ciclo nunca termina, ya que de esta manera se crearían infinitos procesos, con un orden de 2^n, donde n va a ir a infinito.

Luego de realizar esto, se presentó el problema del barbero dormilón, el cuál consistía en una sincronización de procesos. Esto simulaba las siguientes situaciones:

* La barbería consiste en dos habitaciones: la sala de espera con n sillas y la sala del barbero con una sola silla para cortar.
* Cuando no hay clientes el barbero se pone a dormir.
* Cuando el cliente llega, si la sala esta full se va (se pierde un cliente). Si hay lugar para esperar entonces ingresa, si el barbero está durmiendo lo despierta para que le corte el pelo, si el barbero está ocupado entonces el cliente espera en una silla.

Para simular esta situación, se implementó un programa escrito en C, el cual consistía en intercambiar entre los threads *“barbero”*, y *“cliente”*, usando las variables compartidas *“cant\_clientes”*, que representa la cantidad de clientes que están en espera, y *“cant\_sillas”*, el cual indica la cantidad de sillas disponibles dentro de la sala. Por otro lado, se utilizaron dos semáforos, uno para el barbero y otro para el cliente, los cuales permitían la ejecución de los hilos en el orden deseado, y además un mutex llamado *“mi\_mutex”*, el cual cumplía la función de guardar la zona crítica de cada thread, con el fin de que no haya una inconsistencia en los datos.

Por otro lado, se implementó en un programa el diálogo del barbero de Sevilla de la ópera de Gioachino Rossini. Esto se logró a través de threads, los cuales simbolizan cada verso del diálogo *(“Fígaro”)*,y varios semáforos, los cuales se encargan de que el diálogo se muestre en el orden que se desea. El software imprime en pantalla el diálogo del barbero, además de repetirlo las veces que desee el usuario.

Por último, se comprobó el potencial del paralelismo a través de la evaluación del tiempo y velocidad de descarga de un archivo a través de dos métodos de descarga: Por un lado, se utilizó la “forma tradicional” a través de la herramienta *wget*, la cual descarga utilizando un único hilo; y por el otro, la herramienta *axel*, la cual descarga a través de múltiples hilos.

Se compararon los tiempos de descarga de un archivo, el cual pesaba aproximadamente 300 MB, con las herramientas previamente mencionadas. Luego de realizar las mismas, se observó que la herramienta *wget* descarga con mayor velocidad que el axel, pero el axel a su vez, mientras más threads (hilos o procesos) le asignes, mayor velocidad de descarga va a tener. Por eso, cabe la posibilidad de que, si se realiza la misma comparación en una computadora con más recursos, la herramienta *axel* sobrepase a la de *wget*.

El tipo de paralelismo que emplea axel es el paralelismo de datos. Se descarta que sea a través del paralelismo de tareas, porque al ejecutarse la herramienta, en vez de utilizar procesos para diversas tareas independientes dentro de un mismo dato, se observa en la terminal que los procesos creados comparten la misma tarea de descargar partes distribuidas del elemento que se desea adquirir.

A la hora de realizar estas actividades, se presentaron varios obstáculos, para los cuales se plantearon soluciones:

* Hubo un problema al implementar los threads del barbero de Sevilla en orden, debido a que los hilos no se ejecutaban en paralelo y en el orden correcto. Para solucionar ese problema, se optó por utilizar un semáforo por cada thread, además de tres semáforos adicionales, los cuales cumplían con la función de restringir la ejecución del thread 5 hasta que se hayan ejecutado primero los threads 2, 3 y 4.
* El principal problema que encontramos en este ejercicio fue que tomamos por hecho al principio que al final del ciclo, tendríamos solo 2^n veces *“¡Hola mundo!”*. Luego entendimos que en realidad tendríamos la suma de 2^n, 2^n – 1 ... 2^1, ya que por cada iteración se va imprimiendo el mensaje.
* Surgió una dificultad a la hora de implementar el barbero dormilón, la cual era que el programa entraba en bucle. Esto se veía porque a la hora de imprimir en la terminal, se observaba que cada vez que el barbero cortaba el pelo, entraba un cliente infinitamente, por ende nunca se llenaba la sala, ni tampoco quedaba vacía. Debido a esto, se pensó y se implementó otro código que recibe un límite de clientes, y cuando se alcanza corta el ciclo. Pero tuvimos un problema, el cual consistía en que el proceso quedaba en “zombie”.

**Código fuente**

**Supermenú**

#!/bin/bash

#------------------------------------------------------

# PALETA DE COLORES

#------------------------------------------------------

#setaf para color de letras/setab: color de fondo

red=`tput setaf 1`;

green=`tput setaf 2`;

blue=`tput setaf 4`;

bg\_blue=`tput setab 4`;

reset=`tput sgr0`;

bold=`tput setaf bold`;

#------------------------------------------------------

# TODO: Completar con su path

#------------------------------------------------------

proyectoActual=$(pwd);

#------------------------------------------------------

# DISPLAY MENU

#------------------------------------------------------

imprimir\_menu () {

imprimir\_encabezado "\t S U P E R - M E N U ";

echo -e "\t\t El proyecto actual es:";

echo -e "\t\t $proyectoActual";

echo -e "\t\t";

echo -e "\t\t Opciones:";

echo "";

echo -e "\t\t\t a. Ver estado del proyecto";

echo -e "\t\t\t b. Guardar cambios";

echo -e "\t\t\t c. Actualizar repo";

echo -e "\t\t\t f. Abrir en terminal";

echo -e "\t\t\t g. Abrir en carpeta";

echo -e "\t\t\t h. Crear 100 carpetas";

echo -e "\t\t\t i. Borar las 100 carpetas";

echo -e "\t\t\t j. Estados de un Proceso";

echo -e "\t\t\t k. El barbero dormilon";

echo -e "\t\t\t m. La opera de Gioachino Rossini";

echo -e "\t\t\t q. Salir";

echo "";

echo -e "Escriba la opción y presione ENTER";

}

#------------------------------------------------------

# FUNCIONES AUXILIARES

#------------------------------------------------------

imprimir\_encabezado () {

clear;

#Se le agrega formato a la fecha que muestra

#Se agrega variable $USER para ver que usuario está ejecutando

echo -e "`date +"%d-%m-%Y %T" `\t\t\t\t\t USERNAME:$USER";

echo "";

#Se agregan colores a encabezado

echo -e "\t\t ${bg\_blue} ${red} ${bold}--------------------------------------\t${reset}";

echo -e "\t\t ${bold}${bg\_blue}${red}$1\t\t${reset}";

echo -e "\t\t ${bg\_blue}${red} ${bold} --------------------------------------\t${reset}";

echo "";

}

esperar () {

echo "";

echo -e "Presione enter para continuar";

read ENTER ;

}

malaEleccion () {

echo -e "Selección Inválida ..." ;

}

decidir () {

echo $1;

while true; do

echo "desea ejecutar? (s/n)";

read respuesta;

case $respuesta in

[Nn]\* ) break;;

[Ss]\* ) eval $1

break;;

\* ) echo "Por favor tipear S/s ó N/n.";;

esac

done

}

#------------------------------------------------------

# FUNCIONES del MENU

#------------------------------------------------------

a\_funcion () {

imprimir\_encabezado "\tOpción a. Ver estado del proyecto";

echo "---------------------------"

echo "Somthing to commit?"

decidir "cd $proyectoActual; git status";

echo "---------------------------"

echo "Incoming changes (need a pull)?"

decidir "cd $proyectoActual; git fetch origin"

decidir "cd $proyectoActual; git log HEAD..origin/master --oneline"

}

b\_funcion () {

git config --global user.name "Giovanni Sia"

git config --global user.email "giovanni\_sia@yahoo.com"

imprimir\_encabezado "\tOpción b. Guardar cambios";

git init

git remote add origin <https://gitlab.com/Giovanni_S/tp-1.git>

git push -u origin master

decidir "cd $proyectoActual; git add -A";

echo "Ingrese mensaje para el commit:";

read mensaje;

decidir "cd $proyectoActual; git commit -m \"$mensaje\"";

decidir "cd $proyectoActual; git push";

}

c\_funcion () {

imprimir\_encabezado "\tOpción c. Actualizar repo";

decidir "cd $proyectoActual; git pull";

}

f\_funcion () {

imprimir\_encabezado "\tOpción f. Abrir en terminal";

decidir "cd $proyectoActual; xterm &";

}

g\_funcion () {

imprimir\_encabezado "\tOpción g. Abrir en carpeta";

decidir "gnome-open $proyectoActual &";

}

#------------------------------------------------------

# TODO: Completar con el resto de ejercicios del TP, una funcion por cada item

#------------------------------------------------------

h\_funcion () {

imprimir\_encabezado "\tOpción h. Crear 100 carpetas";

for i in {0..99}

do

#Creo las carpetas

mkdir grupo$i

chmod 640 grupo$i

done

}

i\_funcion () {

imprimir\_encabezado "\tOpción i. Borrar las 100 carpetas";

for i in {0..99}

do

cd $proyectoActual

rmdir grupo$i

done

}

j\_funcion(){

imprimir\_encabezado "\tOpción j. Estados de un Proceso";

./Estados\_De\_Un\_Proceso;

}

k\_funcion(){

imprimir\_encabezado "\tOpción k. El barbero dormilon";

./El\_Barbero\_Dormilon

}

m\_funcion(){

imprimir\_encabezado "\tOpción m. La opera de Gioachino Rossini";

./figaro;

}

#------------------------------------------------------

# LOGICA PRINCIPAL

#------------------------------------------------------

while true

do

# 1. mostrar el menu

imprimir\_menu;

# 2. leer la opcion del usuario

read opcion;

case $opcion in

a|A) a\_funcion;;

b|B) b\_funcion;;

c|C) c\_funcion;;

d|D) d\_funcion;;

e|E) e\_funcion;;

f|F) f\_funcion;;

g|G) g\_funcion;;

h|H) h\_funcion;;

i|I) i\_funcion;;

j|J) j\_funcion;;

k|K) k\_funcion;;

m|M) m\_funcion;;

q|Q) break;;

\*) malaEleccion;;

esac

esperar;

done

**Estados de un Proceso**

#include <stdio.h>

#include <limits.h>

int main() {

int valor\_max;

printf("Digite un numero: ");

scanf("%i",&valor\_max);

///////////////////////////////////////////////////////////////

//\*en este punto el proceso esta en Estado Bloqueado (S+D)///// ///////////////////////////////////////////////////////////////

int i=1;

while(i<=valor\_max){

////////////////////////////////////////////////////////////

//en este punto el proceso esta en Estado de Ejecucion (R)// ////////////////////////////////////////////////////////////

printf("%i.\n",i);

i++;

}

return 0;

//////////////////////////////////////////////////////////

//en este punto el proceso esta en Estado Terminado (T)///

//////////////////////////////////////////////////////////

}

/\*

Compilar:

gcc ejercicio2.c -o ejecutar

Ejecutar:

./ejecutar

Ver Proceso con mas opciones y colores:

htop

\*/

**Exclusión Mutua: Barbero Dormilón**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <string.h>

#include <unistd.h>

#include <semaphore.h>

#include <stdbool.h>

sem\_t barberListo;

sem\_t cliente;

sem\_t inicio;

pthread\_mutex\_t mi\_mutex;

int cant\_clientes;

int sillas;

bool barbero\_durmiendo;

void\* Cliente(){

while(true){

sleep(0.000001);

pthread\_mutex\_lock(&mi\_mutex);

if (barbero\_durmiendo && cant\_clientes == 0){

printf("Cliente: cof cof despierte seÃ±or \n");

barbero\_durmiendo = false;

cant\_clientes++;

printf("...le indica como se quiere cortar...-\n");

pthread\_mutex\_unlock(&mi\_mutex);

sem\_post(&cliente);

}else{

pthread\_mutex\_unlock(&mi\_mutex);

//Acceso exclusivo a cant\_clientes

pthread\_mutex\_lock(&mi\_mutex);

if (cant\_clientes <= sillas){

cant\_clientes++;

printf("...llega un cliente y se sienta...\n");

pthread\_mutex\_unlock(&mi\_mutex);

sem\_wait(&barberListo);

printf("Cliente: Ya me toca!!!");

printf("...le indica como se quiere cortar...-\n");

sem\_post(&cliente);

}else{

//Se va si no hay lugar

pthread\_mutex\_unlock(&mi\_mutex);

printf("Cliente(afuera): Esta re lleno, me vooy!!!\n");

}

}

}

}

void\* Barber(){

while(true){

pthread\_mutex\_lock(&mi\_mutex);

if (cant\_clientes == 0 && !barbero\_durmiendo){

barbero\_durmiendo = true;

printf("Barbero: ZZZZZ\n");

}

pthread\_mutex\_unlock(&mi\_mutex);

sem\_wait(&cliente);

pthread\_mutex\_lock(&mi\_mutex);

//Un cliente va a ser atendido

cant\_clientes--;

pthread\_mutex\_unlock(&mi\_mutex);

printf("...XXXXXcortandoXXXXX...\n");

sleep(1);

printf("Barbero: Ya termine, quedaste divinx\n");

//barbero esta libre

printf("Barbero: SIGUIENTE!\n");

sem\_post(&barberListo);

}

}

int main(){

printf("Ingrese cuantas sillas hay: ");

scanf("%i",&sillas);

// agregue esto para ver que onda

sem\_init(&barberListo, 0,0);

sem\_init (&cliente, 0,0);

//-------------------------------

pthread\_t p1;

pthread\_t p2;

int rc;

rc = pthread\_create(&p1,NULL,&Barber,NULL);

rc = pthread\_create(&p2,NULL,&Cliente,NULL);

if (rc){

printf("Error: no disponible creacion thread, %d \n", rc);

exit(-1);

}

pthread\_mutex\_destroy(&mi\_mutex);

pthread\_exit(NULL);

}

**Otra implementación**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <string.h>

#include <unistd.h>

#include <semaphore.h>

int sillas;

int limite\_clientes;

sem\_t barbero;

sem\_t cliente;

pthread\_mutex\_t mi\_mutex;

int cant\_clientes=0;

//0 "Durmiendo", 1 "Cortando"

int estadoBarbero = 0;

void\* Cliente () {

pthread\_mutex\_lock (&mi\_mutex);

if (cant\_clientes < sillas) {

cant\_clientes++;

if(estadoBarbero==0){

printf ("Ejem, ejem...,cof, cof...\n");

estadoBarbero=1;

}else{

printf("Leyendo una revista\n");

sleep (0.1);

}

sem\_post (&cliente);

pthread\_mutex\_unlock (&mi\_mutex);

sem\_wait (&barbero);

}else {

printf ("Chau Chau\n");

pthread\_mutex\_unlock (&mi\_mutex);

}

}

void\* Barbero () {

while (1) {

sem\_wait (&cliente);

pthread\_mutex\_lock (&mi\_mutex);

if(cant\_clientes==0){

estadoBarbero=0;

printf("ZZZZZ...\n");

}else{

cant\_clientes--;

printf ("XXXXX\n");

sleep (0.1);

}

sem\_post (&barbero);

pthread\_mutex\_unlock (&mi\_mutex);

}

pthread\_exit(NULL);

}

int main () {

printf("Ingrese cuantas sillas hay: ");

scanf("%i",&sillas);

printf("Ingrese cuanta gente entrara: ");

scanf("%i",&limite\_clientes);

pthread\_mutex\_init ( &mi\_mutex, NULL);

sem\_init(&barbero,0,0);

sem\_init(&cliente,0,1);

pthread\_t t\_barbero;

pthread\_t t\_cliente[limite\_clientes];

//crea y lanzar ambos threads

int rc;

rc=pthread\_create(&t\_barbero,NULL,Barbero,NULL);

for (int i=0;i<limite\_clientes;i++){

sleep(0.1);

rc=pthread\_create(&t\_cliente[i],NULL,Cliente,NULL);

if (rc){

printf("Error:unable to create thread, %d \n", rc);

exit(-1);

}

}

pthread\_mutex\_destroy(&mi\_mutex);

pthread\_exit(NULL);

return(0);

}

**Sincronización: Barbero de Sevilla**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <string.h>

#include <unistd.h>

#include <semaphore.h>

/\*Declaro el semaforo\*/

sem\_t s2;

sem\_t s3;

sem\_t s4;

sem\_t s5;

sem\_t s6;

int cont=0;

/\*Funciones\*/

void\* Thread1(){

printf(" Fiiiigaro");

sem\_post(&s2);

sem\_post(&s3);

sem\_post(&s4);

pthread\_exit(0);

}

void\* Thread2(){

sem\_wait(&s2);

printf("\n Figaro1");

cont++;

if(cont==3)

{

sem\_post(&s5);

}

pthread\_exit(0);

}

void\* Thread3(){

sem\_wait(&s3);

printf("\n Figaro2");

cont++;

if(cont==3){

sem\_post(&s5);

}

pthread\_exit(0);

}

void\* Thread4(){

sem\_wait(&s4);

printf("\n Figaro3");

cont++;

if(cont==3){

sem\_post(&s5);

}

pthread\_exit(0);

}

void\* Thread5(){

sem\_wait(&s5);

printf("\n Figaro Fi");

sem\_post(&s6);

pthread\_exit(0);

}

void\* Thread6(){

sem\_wait(&s6);

printf("\n Figaro Fa \n");

pthread\_exit(0);

}

int main(){

/\*Creamos los threads\*/

pthread\_t t1;

pthread\_t t2;

pthread\_t t3;

pthread\_t t4;

pthread\_t t5;

pthread\_t t6;

/\*Asigno las funciones que utilizaran los threads\*/

pthread\_create(&t1,NULL,Thread1,NULL);

pthread\_create(&t2,NULL,Thread2,NULL);

pthread\_create(&t3,NULL,Thread3,NULL);

pthread\_create(&t4,NULL,Thread4,NULL);

pthread\_create(&t5,NULL,Thread5,NULL);

pthread\_create(&t6,NULL,Thread6,NULL);

/\*Espero la salida de los threads\*/

pthread\_join(t1,NULL);

pthread\_join(t2,NULL);

pthread\_join(t3,NULL);

pthread\_join(t4,NULL);

pthread\_join(t5,NULL);

pthread\_join(t6,NULL);

return 0;

}