**Informe del Proyecto**

**Sistemas Operativos**

Manuel Garat / Bárbara Mattioli

**Sudoku:**

La implementación del validador de sudoku se realiza de dos formas: con procesos y con hilos. La "lógica" de la solución no se altera con cada esquema: en la primera se crean 27 procesos, nueve para verificar las filas, otros nueve para las columnas y los restantes para las sub-grillas; mientras que con la segunda metodología los procesos simplemente se reemplazan por hilos. La matriz (o sudoku) se obtiene de un archivo con una función implementada con un único proceso. Una vez obtenida esta, se procede a realizar la verificación de la misma que consiste en recorrer las filas, columnas y sub-grillas concurrentemente. La gran diferencia radica en cómo se alma-cena y comprueba el resultado obtenido por cada uno. Con hilos, como comparten el espacio de direccionamiento, con una simple variable compartida, "cuidada" con un semáforo mutex para evitar inconsistencias en las escrituras, basta. Sin embargo con procesos, como no sucede lo que con hilos, se recurre al sistema de archivos. La so-lución que se propone consiste en la escritura de cada proceso en su propio archivo. Si bien no nos parece la so-lución más elegante, como cada proceso escribe exactamente un byte (un total de 27B), y además se asegura que estos archivos "temporales" se eliminan al finalizar la ejecución para que el usuario no pueda verlos, no nos parece la peor opción. Una vez finalizado el cómputo de cada proceso, el padre lee de estos archivos para determinar el resultado final.

**Problemas/Limitaciones:**

* La función que llenaba la matriz a partir del archivo "sudoku.txt" funcionaba en algunas computadoras y no en otras. Esta función consistía en leer caracter a caracter hasta el fin de archivo. La manera en que lo detectábamos era lo que producía la inconsistencia. La solución, después de consultar en la práctica, fue leer línea por línea. Creemos que esta solución no presenta problemas que dependan de la arquitectura donde se la ejecute, pero no podemos asegurarlo. Una posible limitación, dependiendo del punto de vista, es que esta nueva función ignora todoss los caracteres a la derecha del texto especificado por el enunciado (números y comas) y por debajo.
* Para designar a cada hilo qué fila/columna/sub-grilla verificar, se le pasaba a la función que ejecutaban un número. Este provocaba warnings en "pthread\_create" aunque el programa funcionara correctamente. Una nueva "solución" resultó en la situación inversa: evitaba los warnings pero no funcionaba. Al no poder solucionar el inconveniente, y recibir confirmación en la práctica de que estos warnings no son problemáticos, se decidió dejarlos en el programa.
* La primera solución con procesos involucraba que cada uno escribiera en el mismo archivo. Aunque funcionaba correctamente, una consulta en la práctica nos aseguró que eso podría no suceder en algún caso en particular. Como no estábamos seguros de cómo solucionarlo con semáforos, decidimos resolverlo con el sistema de archivos, como se explicó previamente.

**Mini shell:**

El núcleo del mini shell es básicamente un intérprete de comando: el usuario ingresa una cadena de caracte-res y el programa lo decodifica para determinar si es un comando válido o no. Si lo es, crea un proceso que se en-carga de su ejecución. Esta metodología es la que emplean, generalmente, los shell de los sistemas operativos. Por separado se implementan las seis funciones que exige el enunciado: crear un directorio es básicamente ejecutar la función de librearía *mkdir*(); crear un archivo es *fopen*(); listar el contenido de un directorio implica obtener al mismo y luego recorrer cada uno de sus elementos, mostrando sus nombres por pantalla; mostrar el contenido de un archivo requiere recorrerlo y mostrarlo. El que presenta un mayor desafío es eliminar un directorio ya que la función *rmdir*() sólo elimina aquellos que no tienen nada dentro. La solución propuesta asemeja al recorrido de un árbol: dado un directorio (un nodo) como argumento, se evalúa si cada elemento dentro de él es un archivo (una hoja), para así eliminarlo con la función *remove*(), o un directorio. En este último caso, se llama recursiva-mente a la misma función con él como parámetro. Una vez que se vuelve de la recursión se procede a eliminarlo con *rmdir*(). De esta forma, para eliminar un directorio, primero se debe remover su sub-árbol de archivos.

El nombre de las funciones que se implementaron son:

* *mkdir*: crear un directorio
* *lsdir*: listar por pantalla el contenido de un directorio
* *rmdir*: eliminar un directorio
* *mkfil*: crear un archivo
* *lsfil*: listar por pantalla el contenido de un archivo
* *help*: muestra una descripción de la función y sintaxis de cada comando
* *exit*: finaliza la ejecución del shell

**Problemas/Limitaciones:**

* Inicialmente no sabíamos cómo integrar la teoría de procesos con el shell ya que no se veía ninguna situación que requiriera de concurrencia. Sin embargo, al consultarlo en la práctica, aprendimos que la razón por la cual se usan procesos en este tipo de programas no radica en la eficiencia del cómputo sino en la seguridad y extensibilidad: si, por ejemplo, hay un problema en *rmdir*() y se genera un ciclo infinito, todo el shell dejaría de responder. En cambio, si al proceso le sucede esto, el padre (el shell), puede terminarlo. En cuan-to a la extensibilidad, si se quiere agregar un nuevo comando, se debería agregar una función al shell y vol-ver a compilarlo. Al implementarlo con procesos, simplemente se debe proporcionar el ejecutable del nuevo comando.
* Una limitación que presenta es que, si el comando que se quiere ejecutar requiere de un parámetro (listar, crear o eliminar un directorio, crear o listar archivo,), por la manera en que implementamos el decodificador (lee de a palabras), el shell va a esperar el parámetro, es decir, si se ejecuta "*lsdir*", la terminal va a esperar a que el usuario ingrese un directorio. Análogamente, aquellos que no requieran parámetros ignoran to-do lo ingresado a continuación del comando.

**Contenido incluido:**

Se incluyen los siguientes códigos fuente:

* *sudokuProcesos.c:* contiene el validador sudoku implementado con procesos
* *sudokuHilos.c:* contiene el validador sudoku implementado con hilos
* *shell.c:* contiene el núcleo del shell (el “decodificador” de comandos)
* *lsdir.c:* contiene la implementación de dicho comando
* *mkdir.c:* contiene la implementación de dicho comando
* *rmdir.c:* contiene la implementación de dicho comando
* *lsfil.c:* contiene la implementación de dicho comando
* *mkfil.c:* contiene la implementación de dicho comando
* *help.c:* contiene la implementación de dicho comando
* *impresoras\_sin\_prioridad.c:* contiene la implementación del ejercicio 1.2.1. a)
* *impresoras\_con\_prioridad.c* contiene la implementación del ejercicio1.2.1. b)
* *asistente.c:* contiene la implementación del ejercicio 1.2.2. a)

También se incluyen los archivos *lista.h* y *lista.c* necesarios para la resolución del ejercicio 1.2.1. b). Estos implementan una lista de enteros simplemente enlazada sin prioridades.

**Comentarios:** en las implementaciones de *impresoras\_sin\_prioridad.c*, *impresoras\_con\_prioridad.c* y *asistente.c* se utilizan algunas funciones como *sleep(), mostrar\_lista(), etc.,* que no son necesarias para el correcto funcionamiento del ejercicio. Sin embargo, su permanencia se debe a que, en nuestra opinión, ayudan a visualizar la situación que están simulando.

1.2

**Impresoras**

El problema de las impresoras con usuarios sin prioridades lo solucionamos a partir de un semáforo inicializado en 2, es decir en la cantidad total de impresoras. Dos usuarios distintos pueden utilizar cualquiera de las dos impresoras al mismo tiempo. Antes de comenzar a utilizar una, uno de estos bloquea el semáforo que representa la impresora. Mientras una de estas está siendo usada por algún usuario, ningún otro la puede usar. Si valor del semáforo es 1, puede usarse la impresora restante, si es cero ambas estan ocupadas. En este último caso, si un usuario quiere usar una, espera hasta que el semáforo sea liberado. Recién cuando un usuario termina de utilizarla desbloquea el semáforo y la impresora puede ser usada nuevamente.

El caso de las impresoras con usuarios con prioridades lo implementamos de la misma manera pero con la diferencia de utilizar una lista en la cual cada usuario que quiere usar una impresora inserta su prioridad. Una impresora puede ser usada por un usuario que tenga la máxima prioridad o la siguiente a esta de las que se encuentran en la lista. Para esto, cada vez que un usuario quiere usar una impresora, se busca en la lista la máxima prioridad (o la siguiente) insertada para evaluar si este puede utilizar la impresora o si hay otro usuario de mayor prioridad esperando. En caso de no tener la suficiente prioridad, este usuario espera hasta que su prioridad sea la máxima o la segunda mayor para poder utilizarla.

**Asistente**

La idea de la implementación de la solución del problema del Asistente de Docencia (AD) y los estudiantes se resume en el uso de semáforos para modelar si hay o no estudiantes, si el asistente está ocupado, si hay sillas disponibles para que los ayudantes esperen a poder consultar y además el comienzo y finalización de una consulta para que ningún otro estudiante consulte en caso de estar el AD ocupado. Se inicializa el semáforo que representa la disponibilidad de las sillas con el total de las sillas, ya que inicialmente asumimos que no hay ningún estudiante esperando a consultar, por lo tanto todas las sillas están disponibles. El semáforo que representa si el AD está ocupado, está inicializado en 1 porque inicialmente está durmiendo. Los demás semáforo estan inicializados en 0 porque, como dicho previamente, la cantidad de estudiantes que necesitan ayuda al comienzo es nula. Cuando llega el primer estudiante, este despierta al AD con liberando el semáforo “consultando” ya que así comienza la consulta y termina con “consulta\_terminada” para que otro pueda consultar, en caso de haber estudiantes esperando. A pesar de querer implementar con un semáforo la situación específica del AD durmiendo, no pudimos, por lo que concluímos en que si no hay estudiantes el AD no realiza consulta.

**Best Practices for Operating System Deployment**

a) Los modelos de sistemas operativos como servicios tienen el objetivo principal de poder actualizar el sistema eficientemente y de manera continua.

En el artículo “Best Practices for Operating System Deployment”, se habla de las prácticas que se emplearon en la empresa Intel para seguir el ritmo de los lanzamientos y actualizaciones de los sistemas operativos mientras se adaptaba al modelo de sistemas operativos como servicios. Estas prácticas se dividen en etapas básicas: la evaluación del nuevo sistema, la preparación de la empresa para su instalación y, por último, su introducción a la empresa. Estas etapas deben realizarse detalladamente, pero en el menor tiempo posible para no disminuir la productividad.

En la primera etapa se estudia el sistema teniendo en cuenta especialmente el rendimiento, la estabilidad y la experiencia del usuario ya que estos ítems afectan la productividad en la empresa. Por otro lado, se evalúan otras cuestiones como la conectividad, la sincronización, las licencias de software, legales, financieras, de privacidad, etc. Por último, se evalúa el sistema de acuerdo a su productividad, costo y beneficio, y seguridad.

La segunda etapa consiste en verificar que se posea lo necesario para integrar y soportar el nuevo sistema. Para esto se la divide en dos partes: preparación operacional y preparación de aplicación. En la primera se comprueba que se tengan a disposición todas las herramientas necesarias para el desarrollo y el soporte de primer y segundo nivel que se requiere. Se debe tener en cuenta que el nuevo software debe estar capacitado para actualizar una PC o ser instalado en nuevas. Además, debe ser posible acceder al nuevo y al viejo sistema en un mismo dispositivo ya que migrar totalmente de un sistema a otro puede llevar meses. La preparación de aplicación consiste en testear el nuevo sistema en los distintos dispositivos. Para ello se utilizan máquinas virtuales ya que aseguran mayor seguridad en caso de producirse fallos, en comparación al testeo directo en PC. Para esto también se utilizan USB drives y compute sticks. Para agilizar esta tarea se prioriza el testeo de las aplicaciones más críticas para la empresa, es decir las que más se necesitan. Una temprana capacitación del personal también agiliza la transición de un sistema a otro, debido a que cuanto mayor tiempo se necesite para que los empleados aprendan a usar el nuevo software correcta y eficientemente, mayor es el tiempo de transición al nuevo sistema.

La tercera etapa, al igual que la anterior, se divide en sub-etepas. Se comienza instalando el software para solo algunos empleados seleccionados, luego se incluyen otros que quieran migrar al nuevo sistema como usuarios tempranos; en la tercera subetapa los usuarios pueden elegir qué sistema usar al iniciar el dispositivo, y en la cuarta y final etapa el nuevo sistema pasa a ser el estándar mientras que el anterior pasa a ser el secundario.

Una herramienta más reciente para instalar un nuevo sistema es dividir en capas el entorno de cómputo, lo que permite hacer cambios en algunas de ellas sin alterar a las demás. Si el sistema operativo se encuentra en una de estas capas, este puede actualizarse sin interrumpir la productividad de la empresa. Esta última técnica permitiría la actualización permanente del sistema operativo.

b) Es importante conocer parte del funcionamiento de una empresa dedicada a la computación ya que probablemente sea el entorno de trabajo de los estudiantes de la carrera. Además, es fundamental tener un conocimiento actualizado sobre las nuevas tecnologías, tener la capacidad de reconocer ventajas y desventajas de las nuevas actualizaciones, poder realizar un análisis completo

de si una nueva tecnología realmente es mejor que la que se tiene y si compensa el trabajo de realizar la transición del software actual al nuevo. Este tipo de prácticas tomadas por Intel podrían ser utilizadas en el futuro profesional de muchos estudiantes.

**Problemas de planificación**

Ver archivo adjunto *planificacion.xlsx*

* La letra E implica que el estado se está Ejecutando. E(I) implica que el procesador I está ejecutando al proceso; L significa que el proceso está en la cola de Listos; B implica que el proceso está Bloqueado por E/S; T implica que el proceso está Terminado.
* Asumimos que el tiempo que lleva computar las transiciones entre los estados de un proceso es des preciable.
* Los encabezados de las columnas, 0 -> 1, 1 -> 2, etc., indican una el período entre tales ciclos. Se lo hizo de esta forma para mantener un esquema similar al que se presenta en la bibliografía de la materia (donde los ciclos se escriben en las líneas entre celdas).
* Una transición marcada en negrita, más ancha que el resto, implica un cambio de más de un estado (B->L, L->E). Se lo distingue porque, como se dijo en la primera nota, al asumir que el tiempo de transición es despreciable, esta información se perdería.

**Códigos fuente:**

**sudokuProcesos.c:**

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
#include <unistd.h>  
#include <string.h>  
#define error\_fork -1  
#define error\_archivo\_incorrecto -2  
int sudoku[9][9];  
// Verifica que el archivo cumpla con las condiciones de formato.  
// NOTA: Un formato válido asume nueve líneas de números intercalados con comas  
int leerDeArchivo(int sudoku[9][9], FILE \* archivo) {  
 int fila = 0, columna = 0, sudokuValido = 1, columnaDeInsercion = 0;  
 int hay\_para\_leer = 1;  
 char linea[18];  
 hay\_para\_leer = fscanf(archivo, "%s",linea);  
 while ( hay\_para\_leer && fila < 9 && sudokuValido && strlen(linea) == 17 ) {  
 while (columna < 17) {  
 // entre la columna 0 y 17 se describe el sudoku  
 if ( columna%2 == 0 ) {  
 // la posición es par -> tiene que ser un dígito  
 sudokuValido = '0' < linea[columna] && linea[columna] <= '9';  
 sudoku[fila][columnaDeInsercion++] = linea[columna] - '0';  
 }  
 else sudokuValido = linea[columna] == ','; // la posición es impar -> tiene que ser una coma  
 columna++;  
 }  
 if ( sudokuValido ) {  
 fila++;  
 columna = 0;  
 columnaDeInsercion = 0;  
 for ( int i = 0; i < 18; i++ ) linea[i] = '\0';  
 fscanf(archivo,"%s",linea);  
 }  
 }  
 return sudokuValido;  
}  
  
void chequearFila(int fila, FILE \* resultados) {  
 // chequea que la fila del sudoku tenga todos los dígitos del 1 al 9.  
 int columna, sudokuValido=1; int filaAux[9] = {0,0,0,0,0,0,0,0,0}; //el arreglo guarda los dígitos de la fila hasta encontrar alguno repetido.  
 while (columna<9 && sudokuValido ) { // si algún hilo encuentra un error, terminal la ejecución  
 sudokuValido = filaAux[sudoku[fila][columna]-1]==0; // se indexa el arreglo con el número leído menos 1 ( arreglo desde 0 a 8 )  
 if (sudokuValido) filaAux[sudoku[fila][columna]-1] = 1; // no está repetido, entonces se marca como leído  
 columna++;  
 }  
 fprintf(resultados,"%c",sudokuValido+'0');  
}  
  
void chequearColumna(int columna, FILE \* resultados) {  
 // chequea que la columna del sudoku tenga todos los dígitos del 1 al 9.  
 int fila=0; int sudokuValido=1; int columnaAux[9] = {0,0,0,0,0,0,0,0,0}; //el arreglo guarda los dígitos de la columna hasta encontrar alguno repetido.  
 while(fila<9 && sudokuValido) { // si algún hilo encuentra un error, terminal la ejecución  
 sudokuValido = columnaAux[sudoku[fila][columna]-1]==0; // se indexa el arreglo con el número leído menos 1 ( arreglo desde 0 a 8 )  
 if (sudokuValido) columnaAux[sudoku[fila][columna]-1] = 1; // no está repetido, entonces se marca como leído  
 fila++;  
 }  
 fprintf(resultados,"%c",sudokuValido+'0');  
}  
  
void chequearSubGrilla(int numero\_subgrilla,FILE \* resultados) {  
 //Verifica que las subgrilla de 3x3 contiene todos los dígitos del 1 al 9.  
 int fila, filaMax, columna, columnaMin, columnaMax;  
 int leidos [9] = {0,0,0,0,0,0,0,0,0};  
 switch(numero\_subgrilla){  
 case 0: fila = 0; columna = 0; break;  
 case 1: fila = 0; columna = 3; break;  
 case 2: fila = 0; columna = 6; break;  
 case 3: fila = 3; columna = 0; break;  
 case 4: fila = 3; columna = 3; break;  
 case 5: fila = 3; columna = 6; break;  
 case 6: fila = 6; columna = 0; break;  
 case 7: fila = 6; columna = 3; break;  
 case 8: fila = 6; columna = 6; break;  
 }  
 filaMax = fila + 3; columnaMax = columna + 3; // subgrillas de 3x3  
 columnaMin = columna; // para resetear la columna en el while  
 int sudokuValido = 1;  
 while ( fila < filaMax && sudokuValido) {  
 while ( columna < columnaMax && sudokuValido) {  
 sudokuValido = leidos[sudoku[fila][columna]-1]==0; // se indexa el arreglo con el número leído menos 1 ( arreglo desde 0 a 8 )  
 if (sudokuValido) leidos[sudoku[fila][columna]-1] = 1; // no está repetido, entonces se marca como leído  
 columna++;  
 }  
 columna = columnaMin;  
 fila++;  
 }  
 fprintf(resultados,"%c",sudokuValido+'0');  
}  
  
void eliminar\_archivos\_creados(int cant\_archivos) {  
 int i = 0;  
 while ( i < cant\_archivos ) {  
 char nombre\_archivo[2] = "";  
 sprintf(nombre\_archivo,"%i",i);  
 remove(nombre\_archivo);  
 i++;  
 }  
}  
  
int main() { // para escribir y despues leer de archivo: Abrir para escritura cerrar y abrir para lectura.  
 FILE \* archivo = fopen("sudoku.txt","r");  
// FILE \* resultados = fopen("resultados.txt","w+"); //archivo donde se comparten los resultados entre los procesos  
 int sudokuValido = 0;  
 if(archivo == NULL) printf("No se encontró el archivo.");  
 else {  
 sudokuValido = leerDeArchivo(sudoku,archivo);  
 fclose(archivo);  
 int j = 0;  
 if ( sudokuValido ) {  
 int i, pid;  
 for (i=0; i<9; i++) {  
 char nombre\_archivo[2] = "";  
 sprintf(nombre\_archivo,"%i",j);  
 FILE \* archivo = fopen(nombre\_archivo,"w");  
 j++;  
 pid = fork();  
 if (pid<0) {  
 fclose(archivo);  
 eliminar\_archivos\_creados(j-1);  
 printf("error en fork");  
 exit(error\_fork);  
 }  
 else  
 if (pid==0) {  
 chequearFila(i,archivo);  
 fclose(archivo);  
 exit(0);  
 }  
 }  
 for (i=0; i<9; i++) {  
 char nombre\_archivo[2] = "";  
 sprintf(nombre\_archivo,"%i",j);  
 FILE \* archivo = fopen(nombre\_archivo,"w");  
 j++;  
 pid = fork();  
 if (pid<0) {  
 fclose(archivo);  
 eliminar\_archivos\_creados(j-1);  
 printf("error en fork");  
 exit(error\_fork);  
 }  
 else  
 if (pid==0) {  
 chequearColumna(i,archivo);  
 fclose(archivo);  
 exit(0);  
 }  
 }  
 for (i=0; i<9; i++) {  
 char nombre\_archivo[2] = "";  
 sprintf(nombre\_archivo,"%i",j);  
 FILE \* archivo = fopen(nombre\_archivo,"w");  
 j++;  
 pid = fork();  
 if (pid<0) {  
 fclose(archivo);  
 eliminar\_archivos\_creados(j-1);  
 printf("error en fork");  
 exit(error\_fork);  
 }  
 else  
 if (pid==0) {  
 chequearSubGrilla(i,archivo);  
 fclose(archivo);  
 exit(0);  
 }  
 }  
 for (i = 0; i < 27; i++) wait(NULL);  
 i = 0; char caracter;  
 while ( i < 27 ) {  
 char nombre\_archivo[2] = "";  
 sprintf(nombre\_archivo,"%i",i);  
 FILE \* archivo = fopen(nombre\_archivo,"r");  
 if ( sudokuValido ) {  
 caracter = fgetc(archivo);  
 sudokuValido = caracter == '1';  
 }  
 fclose(archivo);  
 remove(nombre\_archivo);  
 i++;  
 }  
 if ( sudokuValido ) printf("El sudoku es válido\n");  
 else printf("El sudoku no es válido\n");  
 }  
 else {  
 printf("Formato de archivo incorrecto.");  
 exit(error\_archivo\_incorrecto);  
 }  
 }  
 return 0;  
}

**sudokuHilos.c:**

#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include <semaphore.h>  
#include <string.h>  
#define error\_archivo\_incorrecto -2  
#define error\_hilo -3  
  
int sudokuValido=1;  
sem\_t mutex;  
int sudoku[9][9];  
pthread\_t thread[27];  
  
// Verifica que el archivo cumpla con las condiciones de formato.  
// NOTA: Un formato válido asume nueve líneas de números intercalados con comas  
int leerDeArchivo( FILE \* archivo) { // igual que leerDeArchivo() en sudokuProcesos.c }  
  
void \* chequearFila(int fila) {  
 // chequea que la fila del sudoku tenga todos los dígitos del 1 al 9.  
 int columna, sv=1; int filaAux[9] = {0,0,0,0,0,0,0,0,0}; //el arreglo guarda los dígitos de la fila hasta encontrar alguno repetido.  
 while (columna<9 && sv && sudokuValido) { // si algún hilo encuentra un error, terminal la ejecución  
 sv = filaAux[sudoku[fila][columna]-1]==0; // se indexa el arreglo con el número leído menos 1 ( arreglo desde 0 a 8 )  
 if (sv) filaAux[sudoku[fila][columna]-1] = 1; // no está repetido, entonces se marca como leído  
 columna++;  
 }  
 sem\_wait(&mutex);  
 if (!sv) sudokuValido = sv;  
 sem\_post(&mutex);  
 return NULL;  
}  
  
void \* chequearColumna(int columna) {  
 // chequea que la columna del sudoku tenga todos los dígitos del 1 al 9.  
 int fila=0; int sv=1; int columnaAux[9] = {0,0,0,0,0,0,0,0,0}; //el arreglo guarda los dígitos de la columna hasta encontrar alguno repetido.  
 while(fila<9 && sv && sudokuValido) { // si algún hilo encuentra un error, terminal la ejecución  
 sv = columnaAux[sudoku[fila][columna]-1]==0; // se indexa el arreglo con el número leído menos 1 ( arreglo desde 0 a 8 )  
 if (sv) columnaAux[sudoku[fila][columna]-1] = 1; // no está repetido, entonces se marca como leído  
 fila++;  
 }  
 sem\_wait(&mutex);  
 if (!sv) sudokuValido = sv;  
 sem\_post(&mutex);  
 return NULL;  
}  
  
void \* chequearSubGrilla(int numero\_subgrilla) {  
 //Verifica que las subgrilla de 3x3 contiene todos los dígitos del 1 al 9.  
 int fila, filaMax, columna, columnaMin, columnaMax;  
 int leidos [9] = {0,0,0,0,0,0,0,0,0};  
 switch(numero\_subgrilla){ // mismo case que el de chequearSubGrilla() en sudokuProcesos.c }  
 filaMax = fila + 3; columnaMax = columna + 3; // subgrillas de 3x3  
 columnaMin = columna; // para resetear la columna en el while  
 int sv = 1;  
 while ( fila < filaMax && sv && sudokuValido ) {  
 while ( columna < columnaMax && sv && sudokuValido ) {  
 sv = leidos[sudoku[fila][columna]-1]==0; // se indexa el arreglo con el número leído menos 1 ( arreglo desde 0 a 8 )  
 if (sv) leidos[sudoku[fila][columna]-1] = 1; // no está repetido, entonces se marca como leído  
 columna++;  
 }  
 columna = columnaMin;  
 fila++;  
 }  
 sem\_wait(&mutex);  
 if (!sv) sudokuValido = sv;  
 sem\_post(&mutex);  
 return NULL;  
}  
  
int main() { // para escribir y despues leer de archivo: Abrir para escritura cerrar y abrir para lectura.  
 FILE \* archivo = fopen("sudoku.txt","r");  
 if(archivo == NULL) printf("No se encontró el archivo.");  
 else {  
 sudokuValido = leerDeArchivo(archivo);  
 fclose(archivo);  
 if ( sudokuValido ) {  
 FILE \* resultados = fopen("resultados.txt","w+"); //archivo donde se comparten los resultados entre los procesos. formato: fila/columna/subgrilla  
 fwrite("000",1,3,resultados);  
 rewind(resultados);  
 sem\_init(&mutex,0,1);  
 int i = 0,error = 0, j=0;  
 while(i<9 && !error){  
 error = pthread\_create(&thread[j++],NULL,chequearFila,i);  
 if ( !error ) error = pthread\_create(&thread[j++],NULL,chequearColumna,i);  
 if ( !error ) error = pthread\_create(&thread[j++],NULL,chequearSubGrilla,i);  
 i++;  
 }  
 if ( error ) {  
 printf("Error en la creación de hilos de verificación del sudoku.\n");  
 exit(error\_hilo);  
 }  
 for (i =0; i<27; i++) pthread\_join(thread[i],NULL);  
 if ( sudokuValido ) printf("El sudoku es válido.");  
 else printf("El sudoku no es válido.");  
 remove("resultados.txt");  
 }  
 else {  
 printf("Formato de archivo incorrecto.");  
 exit(error\_archivo\_incorrecto);  
 }  
 }  
 return 0;  
}

**shell.c:**

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
#include <unistd.h>  
#define max 100  
#define error\_fork -1  
#define error\_exec -2  
  
void proceso\_parametro(char comando[max], char parametro[max]) {  
 scanf("%s",parametro);  
 int pid = fork();  
 if ( pid < 0 ) {  
 printf("Error en la creación del proceso que ejecuta el comando.\n");  
 exit(error\_fork);  
 }  
 else {  
 if ( pid > 0 ) wait(NULL);  
 else {  
 // proceso hijo  
 int estado = execl(comando,comando,parametro,NULL);  
 if ( estado == -1 ) {  
 printf("Error en la creación del proceso que ejecuta el comando.\n");  
 exit(error\_exec);  
 }  
 }  
 }  
}  
  
void proceso\_nada(char comando[max]) {  
 int pid = fork();  
 if ( pid < 0 ) {  
 printf("Error en la creación del proceso que ejecuta el comando.\n");  
 exit(error\_fork);  
 }  
 else {  
 if ( pid > 0 ) wait(NULL);  
 else {  
 // proceso hijo  
 int estado = execl(comando,comando,NULL);  
 if ( estado == -1 ) {  
 printf("Error en la creación del proceso que ejecuta el comando.\n");  
 exit(error\_exec);  
 }  
 }  
 }  
}  
  
void limpiar\_strings(char \* s1, char \* s2) {  
 for ( int i = 0; i < max; i++ ) {  
 s1[i] = '\0';  
 s2[i] = '\0';  
 }  
}  
  
int main(int argc, char \* argv[]) {  
 char comando[max] = "";  
 char parametro[max] = "";  
 int terminar = 0;  
 while(!terminar){  
 printf("Ingrese un comando: ");  
 scanf("%s",comando);  
 if ( !strcmp(comando,"exit") ) terminar = 1;  
 else if ( !strcmp(comando,"help") ) proceso\_nada(comando);  
 else {  
 if ( !strcmp(comando,"rmdir") || !strcmp(comando,"lsdir") || !strcmp(comando,"mkdir") || !strcmp(comando,"mkfil") || !strcmp(comando,"lsfil") ) proceso\_parametro(comando,parametro);  
 else printf("No se reconoce a -%s- como un comando.\n",comando);  
 }  
 limpiar\_strings(comando,parametro);  
 }  
 return 0;  
}

**rmdir.c:**

#include <sys/stat.h>  
#include <dirent.h>  
#include <stdio.h>  
#include <string.h>  
#include <unistd.h>  
#include <stdlib.h>  
#define error\_dir -3  
#define error\_arg -4  
#define max 200  
  
// como la función de librería rmdir elimina solo directorios vacios, primero se elimina su contenido.  
void eliminar\_contenido(char \* ruta) {  
 DIR \* dir;  
 struct dirent \* dp;  
 struct stat statbuf;  
 dir = opendir(ruta);  
 if ( dir != NULL ) {  
 while ( (dp = readdir(dir)) != NULL ) { //obtener contenido hasta que no haya nada más que leer  
 //eliminar los directorios . y .. ya que no se necesitan para la recursión  
 if ( strcmp(dp->d\_name,".") && strcmp(dp->d\_name,"..") ) {  
 //preparar la ruta abosluta del elemento leido para poder obtener información de él  
 char ruta\_copia2[max];  
 strcpy(ruta\_copia2,ruta);  
 strcat(ruta\_copia2,"/");  
 strcat(ruta\_copia2,dp->d\_name);  
 stat(ruta\_copia2, &statbuf);  
 if ( S\_ISDIR(statbuf.st\_mode) == 1 ) { // es un directorio  
 //preparar la ruta nueva para la recursión  
 char ruta\_nueva[max] = "";  
 char ruta\_copia[max] = "";  
 strcpy(ruta\_copia,ruta);  
 strcat(ruta\_copia,"/");  
 strcat(ruta\_nueva,dp->d\_name);  
 strcat(ruta\_copia,ruta\_nueva);  
 eliminar\_contenido(ruta\_copia);  
 // el directorio está vacío  
 rmdir(ruta\_copia);  
 }  
 else remove(ruta\_copia2); // es un archivo  
 }  
 }  
 closedir(dir);  
 }  
 else {  
 printf("Error: no se encontró el directorio.\n");  
 exit(error\_dir);  
 }  
}  
  
int main (int argc, char \* argv[]) {  
 if ( argc > 1 ) {  
 eliminar\_contenido(argv[1]);  
 // el contenido del directorio fue eliminado  
 rmdir(argv[1]);  
 }  
 else {  
 printf("Error: no se especificó ningún directorio.\n");  
 exit(error\_arg);  
 }  
 return 0;  
}

**mkfil.c:**

#include <stdio.h>  
#include <stdlib.h>  
#define error\_arg -4  
#define error\_arch -5  
  
// crea un archivo con el nombre especificado por parámetro y lo cierra  
int main(int argc, char\* argv[]){  
 if ( argc > 1 ) {  
 // especifica un archivo  
 FILE \* file = fopen(argv[1],"w");  
 if ( file == NULL ) {  
 printf("Error: no se pudo abrir el archivo.\n");  
 exit(-5);  
 }  
 else fclose(file);  
 }  
 else {  
 printf("Error: no se especificó ningún archivo.\n");  
 exit(error\_arg);  
 }  
 return 0;  
}

**mkdir.c:**

#include <sys/stat.h>  
#include <sys/types.h>  
#include <stdlib.h>  
#include <stdio.h>  
#define error\_dir -3  
#define error\_arg -4  
  
//Crea un directorio nuevo con el nombre especificado por parámetro.  
int main(int argc, char\* argv[]){  
 if ( argc > 1 ) {  
 int resultado = mkdir(argv[1],0777);  
 if ( resultado == -1 ) {  
 printf("Error: no se pudo crear el directorio.\n");  
 exit(error\_dir);  
 }  
 }  
 else {  
 printf("Error: no se especificó ningún archivo.\n");  
 exit(error\_arg);  
 }  
 return 0;  
}

**lsfil.c:**

#include <stdio.h>  
#include <stdlib.h>  
#define error\_arg -4  
#define error\_arch -5  
  
//Muestra en la pantalla el contenido del archivo especificado por parámetro.  
int main (int argc, char \* argv[]){  
 if ( argc > 1 ) {  
 FILE \* file = fopen(argv[1], "r");  
 if ( file == NULL ) {  
 printf("Error: no se pudo abrir el archivo.\n");  
 exit(error\_arch);  
 }  
 else {  
 char c;  
 int finArchivo = fscanf(file, "%c", &c);  
 while ( finArchivo != EOF ) {  
 printf("%c",c);  
 finArchivo = fscanf(file, "%c", &c);  
 }  
 fclose(file);  
 printf("\n");  
 }  
 }  
 else {  
 printf("Error: no se especificó ningún archivo.\n");  
 exit(error\_arg);  
 }  
 return 0;  
}

**lsdir.c:**

#include <sys/types.h>  
#include <dirent.h>  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
#define error\_dir -3  
#define error\_arg -4  
  
// lista el directorio pasado por parámetro  
int main(int argc, char \* argv[]) {  
 DIR \* directorio;  
 if ( argc == 1 ) {  
 printf("Error: no se especificó ningún archivo.\n");  
 exit(error\_arg);  
 }  
 else directorio = opendir(argv[1]); // abrir el directorio argumento  
 if ( directorio != NULL ) {  
 struct dirent \* elem\_dir; // para almacenar los elementos del diretorio  
 elem\_dir = readdir(directorio);  
 int hay\_elementos = elem\_dir!=NULL;  
 while( hay\_elementos ) { // leer hasta que no haya más elementos en el directorio  
 if ( strcmp(elem\_dir->d\_name,".") && strcmp(elem\_dir->d\_name,"..") ) printf("%s\n",elem\_dir->d\_name); // no incluir . ni ..  
 elem\_dir = readdir(directorio);  
 hay\_elementos = elem\_dir!=NULL;  
 }  
 closedir(directorio);  
 }  
 else {  
 printf("Error: no se encontró el directorio.\n");  
 exit(error\_dir);  
 }  
 return 0;  
}

**help.c:**

#include <stdio.h>  
  
// muestra un mensaje por pantalla con la descripción de los comandos.  
int main() {  
 printf("Los comandos que ofrece la terminal son:\n- mkdir [nombre\_directorio]: crea un directorio con el nombre especificado.\n- lsdir [nombre\_directorio]: lista el contenido del directorio especificado por pantalla.\n- rmdir [nombre\_directorio]: elimina el directorio especificado y su contenido.\n- mkfil [nombre\_archivo]: crea un archivo con el nombre especificado.\n- lsfil [nombre\_archivo]: lista el contenido del archivo especificado por pantalla.\n- exit: finaliza la ejecución de la terminal.\n");  
 return 0;  
}

**impresoras\_sin\_prioridad.c:**

#include <stdio.h>  
#include <semaphore.h>  
#include <pthread.h>  
#include <unistd.h>  
#define n 15  
  
sem\_t impresora;  
pthread\_t threads[n];  
  
void \* usuario(int i) {  
 while (1) {  
 printf("Llega el usuario %i.\n",i);  
 sleep(2);  
 sem\_wait(&impresora); // requerir impresora  
 printf("El usuario %i está utilizando una impresora.\n",i);  
 sleep(2);  
 printf("El usuario %i libera una impresora.\n",i);  
 sem\_post(&impresora); // liberar impresora  
 sleep(2);  
 }  
 return 0;  
}  
  
int main() {  
 sem\_init(&impresora,0,2);  
 int i, error=0;  
 while(i<n && !error){

error= pthread\_create(&threads[i],NULL,usuario,i);

i++;

}

if (error) {

printf("Error en la creación de hilos.\n");

exit(error\_hilo);

}  
 for ( i = 0; i < n; i++ ) pthread\_join(threads[i],NULL);  
 return 0;  
}

**impresoras\_con\_prioridad.c:**

#include "lista.c"  
#include <semaphore.h>  
#include <unistd.h>  
#include <pthread.h>  
#define n 3  
  
TLista lista; // la lista donde se almacena a cada usuario que quiere utilizar las impresoras  
pthread\_t thread[n]; // para guardar el id de cada hilo  
sem\_t mutex\_lista, impresoras; // un mutex para el acceso a la lista. el otro semáforo es para el acceso a las impresoras  
int prioridades[n]; // las prioridades de cada usuario: el usuario i tiene la prioridad prioridades[i]  
  
// retorna el primer elemento de la lista. se utiliza para determinar el usuario con mayor prioridad  
int pri() {  
 int menor = lista->primer\_celda->elemento;  
 TCelda movil = lista->primer\_celda;  
 while ( movil != POS\_NULA ) {  
 if ( menor > movil->elemento ) menor = movil->elemento;  
 movil = movil->proxima\_celda;  
 }  
 return menor;  
}  
  
// retorna el segundo menor elemento de la lista. se utiliza para determinar el segundo usuario con  
// mayor prioridad (ya que hay dos impresoras)  
int segu(int primero) {  
 int segundo = lista->primer\_celda->elemento;  
 TCelda movil = lista->primer\_celda;  
 while ( movil != POS\_NULA ) {  
 if ( segundo == primero ) segundo = movil->elemento;  
 if ( segundo > movil->elemento && movil->elemento != primero ) segundo = movil->elemento;  
 movil = movil->proxima\_celda;  
 }  
 return segundo;  
}  
  
// para eliminar aquel usuario cuya prioridad es la pasada por parámetro una vez que utilizó una impresora.  
void eliminar(int prioridad) {  
 TCelda movil = lista->primer\_celda;  
 int encontre = 0;  
 while ( !encontre && movil != POS\_NULA ) {  
 encontre = movil->elemento == prioridad;  
 if ( !encontre ) movil = movil->proxima\_celda;  
 }  
 if ( encontre ) l\_eliminar(lista,movil);  
}  
  
// para mostrar la lista por pantalla y lograr una mejor visualización en la ejecución.  
void mostrar\_lista() {  
 printf("[");  
 TCelda movil = lista->primer\_celda;  
 while ( movil != POS\_NULA ) {  
 if ( movil->proxima\_celda == POS\_NULA ) printf("%i",movil->elemento);  
 else printf("%i-",movil->elemento);  
 movil = movil->proxima\_celda;  
 }  
 printf("]\n");  
}  
  
void \* usuario(int prioridad) {  
 int suficiente\_prioridad;  
 int primero, segundo;  
 while (1) {  
 suficiente\_prioridad = 0;  
 sleep(0.5);  
 sem\_wait(&mutex\_lista);  
 sleep(0.5);  
 printf("Llega el usuario con prioridad %i. La lista de usuarios es: ",prioridad);  
 sleep(0.5);  
 // el usuario quiere utilizar una impresora: se lo agrega a la lista.  
 l\_insertar(lista,POS\_NULA,prioridad);  
 sleep(0.5);  
 mostrar\_lista();  
 sleep(0.5);  
 sem\_post(&mutex\_lista);  
 sleep(0.5);  
 while ( !suficiente\_prioridad ) {  
 sem\_wait(&mutex\_lista);  
 printf("Análisis de prioridad del usuario con prioridad %i:\n",prioridad);  
 sleep(2);  
 primero = pri();  
 sleep(2);  
 // puede utilizar la impresora si es el primero de la lista de prioridades.  
 suficiente\_prioridad = prioridad == primero;  
 sleep(2);  
 if ( suficiente\_prioridad ) printf(" El usuario con prioridad %i está primero. Puede utilizar una impresora.\n",prioridad);  
 else {  
 // puede utilizar la impresora si es el segundo de la lista de prioridades (porque hay dos impresoras).  
 segundo = segu(primero);  
 suficiente\_prioridad = prioridad == segundo;  
 if ( suficiente\_prioridad ) printf(" El usuario con prioridad %i está segundo. Puede utilizar una impresora.\n",prioridad);  
 else printf(" El usuario con prioridad %i no tiene prioridad para usar las impresoras.\n",prioridad);  
 }  
 sleep(2);  
 sem\_post(&mutex\_lista);  
 sleep(2);  
 }  
 // si hay una impresora disponible, el usuario prosigue a utilizarla.  
 sem\_wait(&impresoras);  
 sleep(2);  
 printf("El usuario con prioridad %i está usando una impresora.\n",prioridad);  
 sleep(2);  
 sem\_wait(&mutex\_lista);  
 sleep(2);  
 // el usuario utilizó la impresora. por lo tanto, se lo elimina de la lista de usuarios que quieren utilizarla.  
 eliminar(prioridad);  
 sleep(2);  
 printf("El usuario con prioridad %i libera una impresora. La lista de usuarios es: ",prioridad);  
 sleep(2);  
 mostrar\_lista();  
 sleep(2);  
 sem\_post(&mutex\_lista);  
 sleep(2);  
 sem\_post(&impresoras);  
 sleep(2);  
 }  
 return NULL;  
}  
  
// inicializa las prioridades de cada usuario  
void setear\_prioridades() {  
 for (int i = 0; i < n; i++) prioridades[i] = i; // en este caso, la prioridad se define con el número de usuario  
}  
  
int main() {  
 lista = crear\_lista();  
 sem\_init(&mutex\_lista,0,1);  
 sem\_init(&impresoras,0,2);  
 setear\_prioridades();  
 int i, error = 0;  
while(i<n && !error){  
 error = pthread\_create(&thread[i],NULL,usuario,prioridades[i]); // cada usuario conoce su prioridad  
 i++;  
}  
if ( error ) {  
 printf("Error en la creación de hilos.\n");  
 exit(error\_hilo);

}  
 for ( i = 0; i < n; i++ ) pthread\_join(thread[i],NULL);  
 return 0;  
}

**asistente.c:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define cant\_alumnos 7

#define cant\_sillas 3

sem\_t hay\_estudiantes, sillas, ADocupado, consultando, consulta\_terminada;

void \* AD () {

int imprimir\_dormir = 1;

while (1) {

if ( sem\_trywait(&hay\_estudiantes) == 0 ) {

imprimir\_dormir = 1;

sem\_wait(&consultando);

printf("El AD está atendiendo.\n");

sleep(7);

printf("La consulta termina.\n");

sem\_post(&consulta\_terminada);

}

else {

if ( imprimir\_dormir ) {

printf("El AD duerme.\n");

imprimir\_dormir = 0;

}

}}}

void \* estudiante(int i) {

while (1) {

printf("Llega el estudiante %i.\n",i);

sem\_post(&hay\_estudiantes);

sleep(3);

if ( sem\_trywait(&ADocupado) == 0 ) {

//no hay nadie consultando

sem\_post(&consultando);

printf("El estudiante %i consulta.\n",i);

sem\_wait(&consulta\_terminada);

sleep(5);

printf("El estudiante %i termina de consultar.\n",i);

sem\_post(&ADocupado);

sleep(3);

printf("El estudiante %i se va luego de consultar.\n",i);

}

else {

if ( sem\_trywait(&sillas) == 0 ) {

printf("El estudiante %i se sienta a esperar a consultar.\n",i);

sem\_wait(&ADocupado);

sem\_post(&sillas);

sem\_post(&consultando);

printf("El estudiante %i consulta.\n",i);

sem\_wait(&consulta\_terminada); sleep(5);

printf("El estudiante %i termina de consultar.\n",i);

sem\_post(&ADocupado); sleep(3);

printf("El estudiante %i se va después de haber consultado.\n",i);

}

else printf("El estudiante %i se va porque no había sillas.\n",i);

}

}

}

int main() {

sem\_init(&hay\_estudiantes,0,0);

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

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

sem\_init(&consulta\_terminada,0,0);

sem\_init(&sillas,0,cant\_sillas);

pthread\_t threads[cant\_alumnos+1];

int i, error = 0;

error = pthread\_create(&threads[0],NULL,AD,NULL);  
 while(i<cant\_alumnos && !error){  
 error = pthread\_create(&threads[i],NULL,estudiante,i);  
 i++;  
 }  
 if ( error ) {  
 printf("Error en la creación de hilos.\n");  
 exit(error\_hilo);

}

for ( i = 0; i < cant\_alumnos+1; i++ ) pthread\_join(threads[i],NULL);

return 0;

}