Tarea 02 - Sistemas operativos

REPOSITORIO EN GITHUB: https://github.com/Rengatitos/Taller2-OS

Sección 1: Gestión de Archivos y Directorios Avanzada en Linux

1. Usando comandos de Linux, cree una estructura de directorios con permisos específicos. Asegúrese de:
    - Crear los directorios bajo un árbol estructurado que incluya diferentes niveles de permisos (lectura, escritura, y ejecución).

    - Asignar permisos específicos a usuarios, grupos y otros.

    - ***mkdir tallerADA/publico, mkdir tallerADA/privado, mkdir tallerADA/colaboratorio:***

      Este comando crea los directorios especificados dentro del directorio tallerADA. En este caso, has creado tres subdirectorios: público, privado, y colaboratorio. mkdir significa "make directory", que es utilizado para crear nuevos directorios en el sistema.
    - **chmod 755 tallerADA/publico:**
      chmod es el comando utilizado para cambiar los permisos de un archivo o directorio.
      Los permisos se representan mediante un número de tres dígitos. Cada dígito representa los permisos para el usuario (propietario), el grupo y otros usuarios (en ese orden).
      7: Permisos completos para el propietario (lectura, escritura y ejecución).
      5: Permisos de solo lectura y ejecución para el grupo y otros usuarios.
      Por lo tanto, este comando asigna permisos de lectura, escritura y ejecución al propietario del directorio público, mientras que el grupo y otros usuarios tienen permisos de lectura y ejecución.
    - **chmod 700 tallerADA/privado:**
      Aquí el número 700 establece los permisos del directorio privado:
      7: Permisos completos (lectura, escritura y ejecución) solo para el propietario.
      0: Ningún permiso para el grupo ni otros usuarios.
      asegura que solo el propietario pueda acceder y modificar este directorio, manteniéndolo completamente privado.
    - **chmod 770 tallerADA/colaboratorio:**

Con 770, se establecen los permisos del directorio colaboratorio de la siguiente forma:
7: Permisos completos (lectura, escritura y ejecución) para el propietario.
7: Permisos completos para los usuarios del grupo.
0: Sin permisos para otros usuarios.
Este nivel de permisos permite que tanto el propietario como los miembros del grupo puedan colaborar en el directorio, mientras que otros no pueden acceder a él.

Explicación de los resultados:

- **drwxrwx--- 2 colaboratorio**
  d: Indica que es un directorio.
  rwx: El propietario (les) tiene permisos de lectura (r), escritura (w) y ejecución (x).
  rwx: El grupo tiene los mismos permisos que el propietario (lectura, escritura, ejecución).
  ---: Otros usuarios no tienen ningún tipo de permiso sobre el directorio.
  2: Número de enlaces al directorio.
- **drwx------ 2 privado**:
  d: Es un directorio.
  rwx: El propietario tiene permisos de lectura, escritura y ejecución.
  ---: Ni el grupo ni otros usuarios tienen acceso al directorio.
  Todo lo demás es similar al anterior.
- **drwxr-xr-x 2 publico**:
  d: Es un directorio.
  rwx: El propietario tiene permisos de lectura, escritura y ejecución.
  r-x: El grupo tiene permisos de lectura y ejecución, pero no de escritura.
  r-x: Otros usuarios tienen los mismos permisos que el grupo.

2. Utilizando el comando `find`, localice todos los archivos dentro de su sistema que hayan sido modificados en los últimos 5 días y que tengan una extensión específica (por ejemplo, .txt).

```
./home/les/ballon.txt
./home/les/williams.txt
./home/les/apuntes.txt
./home/les/tallerADA/publico/paratodos.txt
./home/les/tallerADA/privado/contra.txt
./home/les/apuntes2.txt
./home/les/Taller2ADA/proyecto/carpeta1/subcarpeta2/archivo2.txt
./home/les/Taller2ADA/proyecto/carpeta1/subcarpeta1/archivo.txt
./home/les/taller.txt
./home/les/ballon/Marks.txt
find: './root': Permiso denegado
```

3. Programar en C++ un algoritmo que dado el valor de n, este muestre el siguiente patrón. Por ejemplo: n=5
   1
   01
   001
   0001
   00001
   Cree el archivo fuente, edición y compilación a través de comandos de Linux.

```
(base) les@les-virtualbox:~/tallerADA$ nano ejer3.cpp
(base) les@les-virtualbox:~/tallerADA$ g++ ejer3.cpp -o ejercicio3
(base) les@les-virtualbox:~/tallerADA$ ./ejercicio3
Introduce el valor de n: 5
1
01
001
0001
00001
(base) les@les-virtualbox:~/tallerADA$ cat ejercicio3
```

```
les@les-virt
Archivo  Acciones  Editar  Vista  Ayuda
les@les-virtualbox: ~/tallerADA
  GNU nano 6.2
#include <iostream>

int main() {
    int n;

    std::cout << "Introduce el valor de n: ";
    std::cin >> n;

    for (int i = 0; i < n; ++i) {
        // Imprimir ceros
        for (int j = 0; j < i; ++j) {
            std::cout << "0";
        }
        // Imprimir uno
        std::cout << "1" << std::endl;
    }

    return 0;
}
```

4. Muestre en pantalla el contenido del archivo fuente anterior a través de un comando.

```
(base) les@les-virtualbox:~/tallerADA$ cat ejer3.cpp
#include <iostream>

int main() {
    int n;

    std::cout << "Introduce el valor de n: ";
    std::cin >> n;

    for (int i = 0; i < n; ++i) {
        // Imprimir ceros
        for (int j = 0; j < i; ++j) {
            std::cout << "0";
        }
        // Imprimir uno
        std::cout << "1" << std::endl;
    }

    return 0;
}
```

Sección 2: Programación en C para Manipulación de Procesos

5. Escriba un programa en **C** que implemente un sistema de multiprocesamiento con el uso de *fork()*:

```
mezam@Ubuntu:~$ cd Documents
mezam@Ubuntu:~/Documents$ ls
mezam@Ubuntu:~/Documents$ touch SistmMultiproces.c
mezam@Ubuntu:~/Documents$ ls
SistmMultiproces.c
mezam@Ubuntu:~/Documents$ nano SistmMultiproces.c
mezam@Ubuntu:~/Documents$ gcc SistmMultiproces.c -o ejc
mezam@Ubuntu:~/Documents$ ./ejc
```

   - El programa debe crear tres procesos hijos que ejecuten una tarea simple (por ejemplo, contar hasta 1000).

```
mezam@Ubuntu:~/Documents$ ./ejc
Proceso padre (PID: 4509) creando procesos hijos...
Proceso hijo 1 creado con PID: 4510
Proceso hijo 2 creado con PID: 4511
Proceso hijo 1 (PID: 4510): contando hasta 1000...
Proceso 1 (PID: 4510) ha contado hasta 200
Proceso 1 (PID: 4510) ha contado hasta 400
Proceso 1 (PID: 4510) ha contado hasta 600
Proceso 1 (PID: 4510) ha contado hasta 800
Proceso 1 (PID: 4510) ha contado hasta 1000
Proceso hijo 1 (PID: 4510) ha terminado de contar.
Proceso hijo 3 creado con PID: 4512
Proceso hijo 2 (PID: 4511): contando hasta 1000...
```

```
Proceso 2 (PID: 4511) ha contado hasta 1000
Proceso hijo 2 (PID: 4511) ha terminado de contar.
Proceso hijo 3 (PID: 4512): contando hasta 1000...
Proceso 3 (PID: 4512) ha contado hasta 200
Proceso 3 (PID: 4512) ha contado hasta 400
Proceso 3 (PID: 4512) ha contado hasta 600
Proceso 3 (PID: 4512) ha contado hasta 800
Proceso 3 (PID: 4512) ha contado hasta 1000
Proceso hijo 3 (PID: 4512) ha terminado de contar.
Proceso padre (PID: 4509) esperando a que los hijos terminen...
Todos los procesos hijos han terminado.
```

   - Use el comando `ps` para identificar los PID de los procesos creados y muestre el árbol de procesos correspondiente.

```
Mostrando el árbol de procesos con ps -ejH:
    PID    PGID     SID TTY          TIME CMD
      2       0       0 ?        00:00:00 kthreadd
      3       0       0 ?        00:00:00   pool_workqueue_release
      4       0       0 ?        00:00:00   kworker/R-rcu_g
      5       0       0 ?        00:00:00   kworker/R-rcu_p
      6       0       0 ?        00:00:00   kworker/R-slub_
      7       0       0 ?        00:00:00   kworker/R-netns
```

```
  2957    2957    2957 ?        00:00:08     gnome-terminal-
  2965    2965    2965 pts/0    00:00:00       bash
  4509    4509    2965 pts/0    00:00:00         ejc
  4513    4509    2965 pts/0    00:00:00           sh
  4514    4509    2965 pts/0    00:00:00             ps
  4265    4265    4265 pts/1    00:00:00       bash
  3185    1658    1658 ?        00:00:00     snap
  4384    4384    4384 ?        00:00:00   fwupd
```

```
mezam@Ubuntu:~/Documents$ ps -e --forest
    PID TTY          TIME CMD
      2 ?        00:00:00 kthreadd
      3 ?        00:00:00  \_ pool_workqueue_release
      4 ?        00:00:00  \_ kworker/R-rcu_g
      5 ?        00:00:00  \_ kworker/R-rcu_p
      6 ?        00:00:00  \_ kworker/R-slub_
      7 ?        00:00:00  \_ kworker/R-netns
      9 ?        00:00:00  \_ kworker/0:1-events
     10 ?        00:00:00  \_ kworker/0:0H-events_highpri
     12 ?        00:00:00  \_ kworker/R-mm_pe
     13 ?        00:00:00  \_ rcu_tasks_kthread
     14 ?        00:00:00  \_ rcu_tasks_rude_kthread
     15 ?        00:00:00  \_ rcu_tasks_trace_kthread
     16 ?        00:00:00  \_ ksoftirqd/0
```

- Documente todo el flujo de ejecución.

Tabla 1 Descripción del Multiprocesamiento

Proceso	Tiempo Inicio	Tarea
Proceso Padre (PID: 4509)	T0	Crear 3 hijos procesos
Proceso hijo 1 (PID: 4510)	T1	contar números 1 hasta 1000
Proceso hijo 2 (PID: 4511)	T2	contar números 1 hasta 1001
Proceso hijo 3 (PID: 4512)	T3	contar números 1 hasta 1002

Flujo de ejecución:

1. **Proceso padre inicial:**
   - El programa principal (**main**()) comienza su ejecución.
   - El proceso principal (**padre**) obtiene su PID y lo imprime.
   - A continuación, llama a **fork**() tres veces para crear tres procesos hijos.
2. **Creación de procesos hijos:**
   - En cada llamada a fork(), se crea un nuevo proceso hijo.

- Si fork() devuelve un valor de 0, significa que estamos en un proceso hijo, y ese proceso ejecuta la función contar().
- Si fork() devuelve un valor mayor que 0, indica que estamos en el proceso padre, que sigue ejecutando el código y registrando los PIDs de los hijos creados.

3. **Ejecución de la tarea en los hijos:**

   - Cada proceso hijo entra en la función contar(), donde imprime su ID y su PID.
   - Cada hijo cuenta del 1 al 1000, imprimiendo el progreso cada 200 iteraciones.
   - Una vez que el proceso hijo termina de contar, imprime un mensaje indicando que ha terminado y luego sale (exit(0)).

4. **Espera en el padre:**

   - El proceso padre, después de crear los tres hijos, utiliza la función wait() para esperar que cada uno de los hijos termine.
   - Mientras tanto, el padre no realiza ninguna otra tarea hasta que todos sus hijos hayan finalizado.

5. **Mostrar el árbol de procesos:**

   - Una vez que todos los hijos han terminado, el proceso padre ejecuta el comando **ps -ejH** para mostrar el árbol de procesos en el sistema.
   - En este punto, debería mostrar el proceso padre y los tres hijos que fueron creados.

6. **Fin:**

   - Cuando todos los hijos terminan su ejecución, el proceso padre imprime un mensaje final indicando que todos los procesos hijos han finalizado.
   - El programa finaliza correctamente, y todos los procesos creados ya han terminado su ejecución.

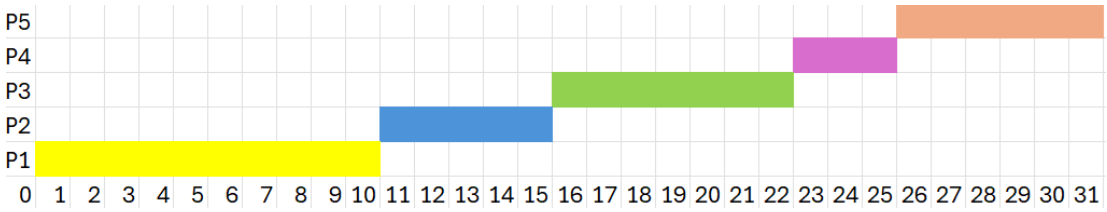
Sección 3: Simulación de Planificación de Procesos

6. Aplique los métodos de planificación de procesos que se han visto en clase. Realice una comparación gráfica del rendimiento de cada algoritmo en términos de **tiempo de espera promedio** y **tiempo de respuesta promedio**. Justifique cuál planificación es mejor y en qué condiciones.

- **FCFS**: Ideal para sistemas simples o por lotes cuyos procesos tienen tiempos de ejecución similares. Los procesos largos hacen esperar a los cortos.

Proceso	T. CPU	T. llegada	T. Inicio	T. Finalización	T. Retorno	T. Espera	INDEX
P1	10	0	0	10	10	0	1
P2	5	1	10	15	14	9	0.357
P3	7	2	15	22	20	13	0.35
P4	3	3	22	25	22	19	0.136
P5	6	4	25	31	27	21	0.222

Gráfico:

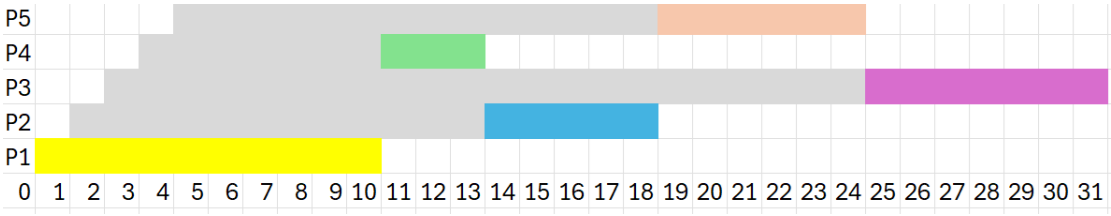


- **FSJF**: Mejor desempeño con tiempos de ejecución previsibles. El objetivo es minimizar el tiempo de espera medio. El algoritmo selecciona aquel proceso cuyo próximo ciclo de ejecución de CP sea menor. El problema está en conocer dichos valores, pero podemos predecirlos usando la información de los ciclos anteriores ejecutados.

.

Proceso	T. CPU	T. llegada	T. Inicio	T. Finalización	T. Retorno	T. Espera	INDEX
P1	10	0	0	10	10	0	1
P2	5	1	13	18	17	12	0.294
P3	7	2	24	31	29	22	0.241
P4	3	3	10	13	10	7	0.3
P5	6	4	18	24	20	14	0.3

Gráfico:



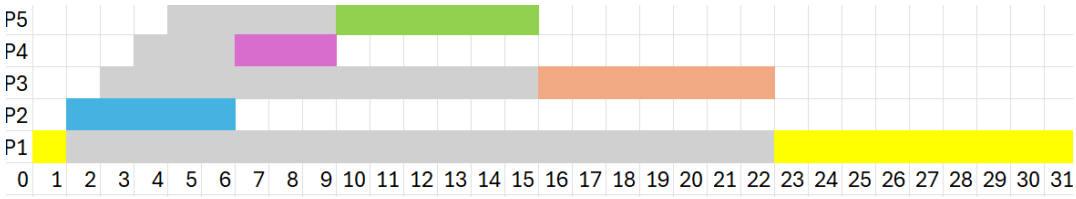
- **SRTF**: Adecuado cuando los procesos cortos son prioritarios y el sistema requiere un tiempo de respuesta bajo. Problemas de inanición: los trabajos largos no se ejecutarán mientras haya trabajos cortos.
-

Proceso	T. CPU	T. llegada	T. Inicio	T. Finalización	T. Retorno	T. Espera	INDEX
P1	10	0	0 ; 22	1 ; 31	31	21	0.323
P2	5	1	1	6	5	0	1
P3	7	2	15	22	20	13	0.35
P4	3	3	6	9	6	3	0.5
P5	6	4	9	15	11	5	0.545

Cola:

	10	5	3	6	7	9
Q	P1	P2	P4	P5	P3	P1
	1	2	3	4	5	6

Gráfico:



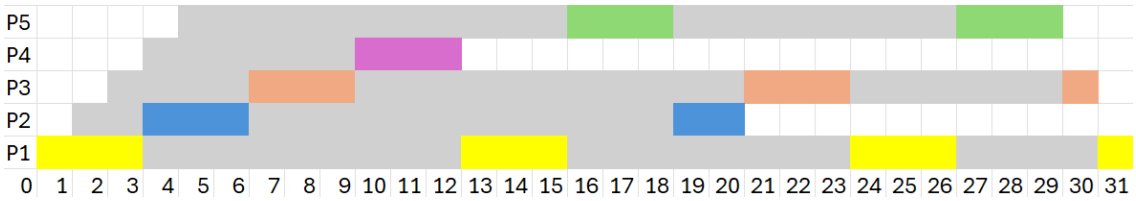
- **Round Robin**: Es el mejor para sistemas interactivos o en tiempo compartido, donde todos los procesos deben tener una oportunidad; si el proceso se bloquea o termina antes de agotar su quantum también se alterna el uso de la CPU.
  q=3

Proceso	T. CPU	T. llegada	T. Inicio	T. Finalización	T. Retorno	T. Espera	INDEX
P1	10	0	0; 12;23;30	3;15;26;31	31	21	0.323
P2	5	1	3;18	6;20	19	14	0.263
P3	7	2	6;20	9;23;30	28	21	0.25
P4	3	3	9	12	9	6	0.33
P5	6	4	15;26	18;29	25	19	0.24

Cola:

	10	5	7	3	7	6	2	4	4	3	1	1
Q	P1	P2	P3	P4	P1	P5	P2	P3	P1	P5	P3	P1
	1	2	3	4	5	6	7	8	9	10	11	12

Gráfico:



CONCLUSIÓN:
El **algoritmo Round Robin** es el más confiable para sistemas interactivos, ya que garantiza que todos los procesos reciban tiempo de CPU de manera equitativa y constante. Esto lo convierte en una excelente opción cuando el objetivo es mantener un buen tiempo de respuesta en sistemas donde los usuarios esperan interacción rápida.
.
En definitiva, la mejor opción dependerá del tipo de sistema y de las condiciones particulares de uso.

Sección 4: Saturación de Procesadores con Hilos en C

7. Escriba un programa en **C** que cree un número de hilos igual a la cantidad de núcleos de su sistema. Cada hilo debe realizar una operación que simule la carga del procesador (por ejemplo, un bucle que dure al menos 120 segundos). Asegúrese de:
    - Mostrar el ID de cada hilo en pantalla.

```
mezam@Ubuntu:~/Documents$ ./NumHilosIgualNumNucleos
Creando 2 hilos, uno por cada núcleo del sistema...
Hilo ID: 0, PID del Hilo: 5691
Hilo ID: 1, PID del Hilo: 5692
Todos los hilos han terminado.
```

    - Documentar cómo el sistema maneja la asignación de hilos en diferentes núcleos.
    -

CODIGO

1. Función simular_carga():
Esta función se encarga de la operación que realiza cada hilo. Dentro de ella, cada hilo:
- Obtiene su PID usando syscall(SYS_gettid).
- Imprime su ID y su PID.
- Simula carga de trabajo con un bucle que cuenta hasta un valor muy grande (1e10).

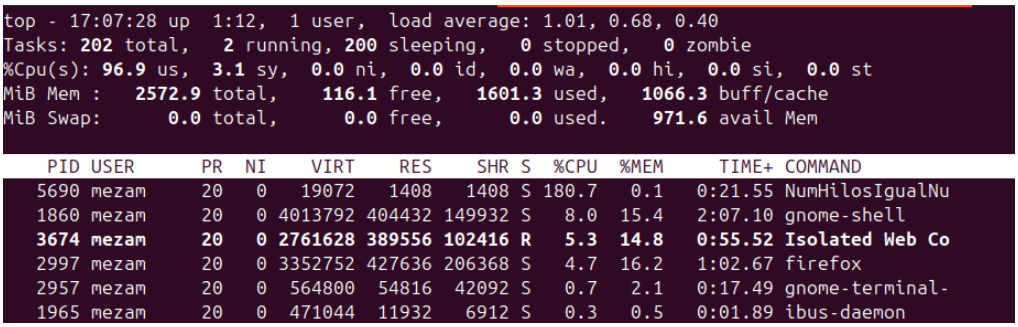
2. Función main():

- Obtiene el número de núcleos del sistema con get_nprocs().

- Crea un hilo por cada núcleo con pthread_create(), asignando la función simular_carga() a cada hilo.
- Espera a que todos los hilos terminen usando pthread_join() para evitar que el programa principal termine antes que los hilos.

8. **Extras**: Mida el uso de CPU de cada hilo durante la ejecución usando comandos como `top` o `htop`, y analice el impacto en el rendimiento del sistema.

PID NumHilosIgualNumNucleos: 5690



ASIGNACIÓN DE HILOS EN DIFERENTES NÚCLEOS

1. Creación de Hilos
Cuando creas hilos con pthread_create(), el sistema no asigna de inmediato los hilos a núcleos específicos. Los coloca en una cola de tareas.

2. Distribución Automática
El **scheduler** decide en qué núcleo ejecuta cada hilo, basándose en la carga de trabajo:

- Si un núcleo está ocupado, moverá el hilo a otro núcleo menos ocupado.
- Trata de balancear la carga entre todos los núcleos, como ejemplo si hay 3 núcleos asignara un hilo a cada núcleo.

3. Migración de Hilos
El sistema puede mover hilos entre núcleos durante la ejecución si detecta que un núcleo está sobrecargado o si otros tienen menos trabajo.

4. Monitoreo
Puedes ver cómo los hilos se distribuyen entre los núcleos usando comandos como htop o top -H, donde puedes observar el uso de CPU de cada hilo.

Entrega:

- Subir el código y scripts a un repositorio en GitHub (opcional) y proporcionar un informe en PDF con las explicaciones de cada ejercicio.