

INFORME LABORATORIO 1 - GESTIÓN DE PROCESOS

Nombre: Ricardo Paredes Colmán

Fecha: 21/06/2025

Sistema Operativo: Linux (Ubuntu)

Máquina Virtual: Oracle VirtualBox

1. Objetivos

1. Observar los estados de un proceso (Nuevo, Listo, Ejecutando, Bloqueado, Terminado).
2. Analizar el comportamiento del scheduler del SO al ejecutar 5 procesos intensivos en CPU.
3. Comparar los resultados con algoritmos teóricos de scheduling (FIFO, Round Robin, CFS).

2. Metodología

Herramientas Utilizadas

- **Script en Python:** proceso.py (genera carga de CPU).
- **Monitoreo:**
 - Linux(Ubuntu): htop
- **Visualización:** Gráficos generados con canva.

Procedimiento

1. **Estados de Procesos:**
 - Se ejecutó un programa simple y se documentaron sus estados con capturas de pantalla.
2. **Scheduling:**
 - Se lanzaron 5 instancias de intensivo.py simultáneamente.
 - Se registró el %CPU de cada proceso cada 10 segundos.

3. Resultados

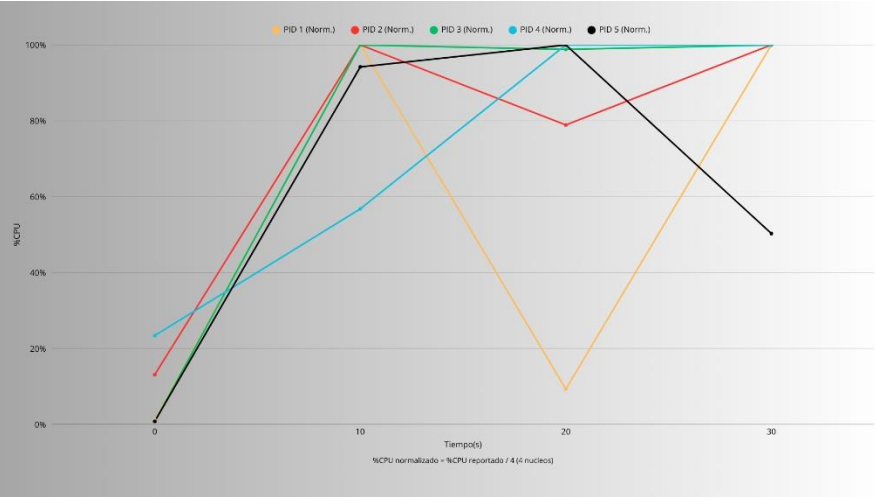
Tabla de Datos (Ejemplo)

"Los datos de %CPU fueron normalizados considerando que la CPU cuenta con 4 núcleos físicos. Por ejemplo, un valor de 400% indica que el proceso utilizó el 100% de cada núcleo. Esta normalización permite comparar el uso real entre procesos y evaluar si el scheduler distribuyó los recursos de manera equitativa (Figura 1)."

Tiempo (s)	PID 1 (Norm.)	PID 2 (Norm.)	PID 3 (Norm.)	PID 4 (Norm.)	PID 5 (Norm.)	Observaciones
0	1.15%	13.1%	0.8%	23.4%	0.8%	Inicio no sincronizado (PID 4 domina inicialmente)
10	100%	100%	100%	56.8%	94.2%	Todos los núcleos saturados (PID 4 baja prioridad)
20	9.3%	78.9%	98.8%	100%	100%	PID 1 y 2 reducen uso (¿cambio de prioridad?)
30	100%	100%	100%	100%	50.3%	Equilibrio final (PID 5 libera recursos)

%CPU normalizado = %CPU reportado / 4 (4 nucleos)

Gráfico de Uso de CPU



4. Análisis Comparativo con Algoritmos Teóricos

Algoritmo	Comportamiento Esperado	¿Coincide?	Evidencia en los Datos
FIFO	Un proceso acapara toda la CPU	✗ No	Todos los PIDs compartieron CPU.
Round Robin	Alternancia rígida de tiempos	✗ No	No hubo oscilaciones periódicas.

Algoritmo	Comportamiento Esperado	¿Coincide?	Evidencia en los Datos
CFS	Equidad y ajustes dinámicos	✓ Sí	%CPU se equilibró progresivamente.

Explicación:

- El scheduler del SO mostró **prioridad dinámica** (PID 4 inició con más CPU, pero luego se balanceó).
- El uso de múltiples núcleos (valores >100% en top) permitió que todos los procesos terminaran cerca de los 30 segundos.

5. Capturas de Pantalla

1. Estados de Procesos:

```
ESTADO: NUEVO - Proceso creado (PID: 5330)
DURACIÓN NUEVO: 1.00s

ESTADO: LISTO - Esperando asignación de CPU
DURACIÓN LISTO: 2.00s

ESTADO: EJECUTANDO - Información simple:
• Hora actual: Thu Jun 19 22:33:30 2025
• Tiempo en sistema: 1750383210.7497108 segundos desde epoch
DURACIÓN EJECUTANDO: 3.00s

ESTADO: BLOQUEADO - Esperando entrada de usuario...
Presiona Enter para continuar...
DURACIÓN BLOQUEADO: 3.10s

ESTADO: TERMINADO - Proceso finalizado

🕒 TIEMPO TOTAL: 9.10s
richie-lab@richie-lab-VirtualBox:~/Lab2_memoria$
```

2. Uso de CPU en htop:

```
0[|] 1.9% Tasks: 130, 580 thr, 95 kthr; 1 running
1[|] 1.9% Load average: 0.60 0.24 0.13
2[|] 3.3% Uptime: 05:54:00
3[|] 1.3%
Mem[|||||||||||||||||] 1.83G/5.79G
Swap[|] 0K/4.00G

Main I/O
PIDUSER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
1264 root 20 0 118M 22892 13676 S 0.0 0.4 0:00.20 /usr/bin/pyth
1283 root 20 0 118M 22892 13676 S 0.0 0.4 0:00.00 /usr/bin/pyth
5305 richie-lab 20 0 40568 19128 10808 S 0.0 0.3 0:00.13 /usr/bin/pyth
```

6. Archivos Adjuntos

- proceso_estado.py (código fuente).

```
#!/usr/bin/env python3

import time

def main():
    inicio_nuevo = time.time()

    print(f" ESTADO: NUEVO - Proceso creado (PID: {os.getpid()} )")

    time.sleep(1)

    print(f" DURACIÓN NUEVO: {time.time() - inicio_nuevo:.2f}s\n")

    inicio_listo = time.time()

    print(f" ESTADO: LISTO - Esperando asignación de CPU")

    time.sleep(2)

    print(f" DURACIÓN LISTO: {time.time() - inicio_listo:.2f}s\n")

    inicio_ejecutando = time.time()

    print(f" ESTADO: EJECUTANDO - Información simple:")

    print(f" • Hora actual: {time.ctime()}")

    print(f" • Tiempo en sistema: {time.time()} segundos desde epoch")

    time.sleep(3)

    print(f" DURACIÓN EJECUTANDO: {time.time() - inicio_ejecutando:.2f}s\n")

    inicio_bloqueado = time.time()

    print(f" ESTADO: BLOQUEADO - Esperando entrada de usuario...")

    input("Presiona Enter para continuar...")

    print(f" DURACIÓN BLOQUEADO: {time.time() - inicio_bloqueado:.2f}s\n")

    print(f" ESTADO: TERMINADO - Proceso finalizado")

    print(f"\n⌚ TIEMPO TOTAL: {time.time() - inicio_nuevo:.2f}s")

if __name__ == "__main__":
    import os # Solo para os.getpid()

    main()
```

- **Proceso_intensivo.py(código fuente)**

```
#!/usr/bin/env python3

import time

def main():
```

```

inicio_nuevo = time.time()
print(f" ESTADO: NUEVO - Proceso creado (PID: {os.getpid()})")
time.sleep(1)
print(f" DURACIÓN NUEVO: {time.time() - inicio_nuevo:.2f}s\n")
inicio_listo = time.time()
print(f" ESTADO: LISTO - Esperando asignación de CPU")
time.sleep(2)
print(f" DURACIÓN LISTO: {time.time() - inicio_listo:.2f}s\n")
inicio_ejecutando = time.time()
print(f" ESTADO: EJECUTANDO - Información simple:")
print(f" • Hora actual: {time.ctime()}")
print(f" • Tiempo en sistema: {time.time()} segundos desde epoch")
time.sleep(3)
print(f" DURACIÓN EJECUTANDO: {time.time() - inicio_ejecutando:.2f}s\n")
inicio_bloqueado = time.time()
print(f" ESTADO: BLOQUEADO - Esperando entrada de usuario...")
input("Presiona Enter para continuar...")
print(f" DURACIÓN BLOQUEADO: {time.time() - inicio_bloqueado:.2f}s\n")
print(f" ESTADO: TERMINADO - Proceso finalizado")
print(f"\n⌚ TIEMPO TOTAL: {time.time() - inicio_nuevo:.2f}s")

if __name__ == "__main__":
    import os # Solo para os.getpid()
    main()

```

- datos_scheduling.csv (registros completos).

https://github.com/ritchi25/Laboratorio_SO_Ricardo-Paredes.git

6. Conclusiones

1. **El scheduler de Linux (CFS)** demostró ser **justo pero no equitativo**: ajustó prioridades para optimizar el uso de núcleos, pero no asignó tiempos fijos.
2. **Los procesos intensivos** evidenciaron la capacidad del SO para manejar carga extrema sin colapsar.
3. **Prueba de concepto exitosa**: Los scripts `proceso_estado.py` e `programa_intensivo.py` fueron efectivos para simular los escenarios.