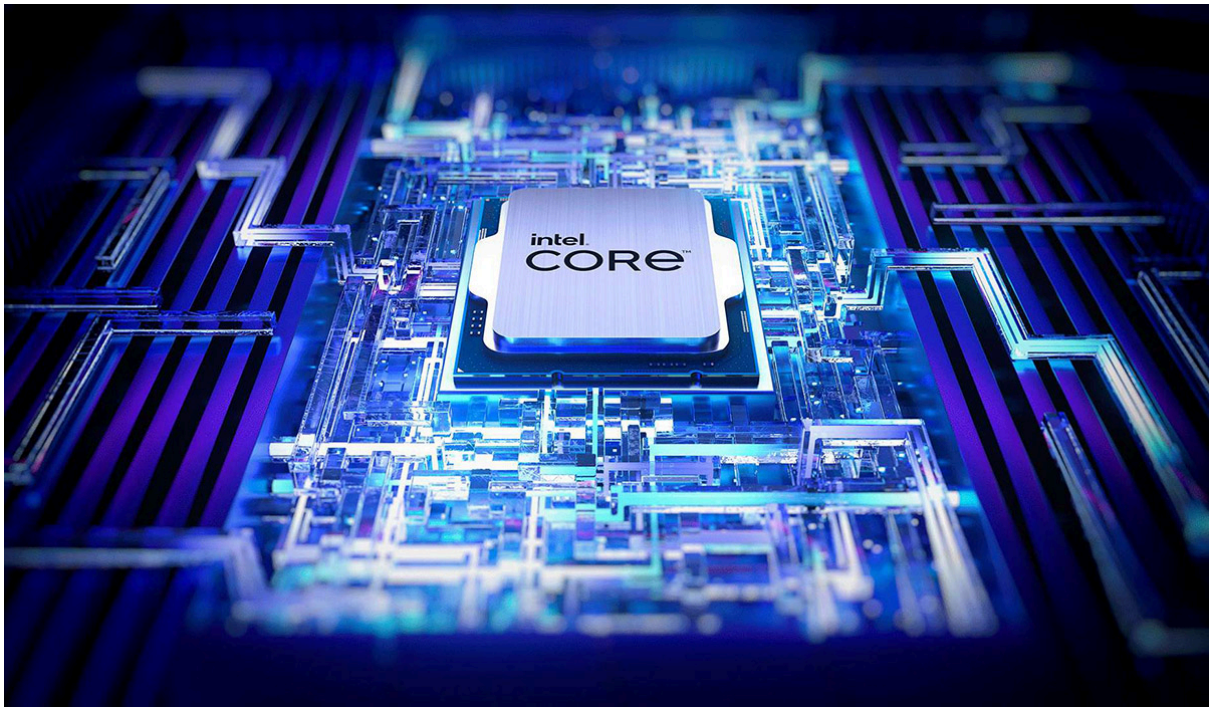


2.3- Optimización de procesos multi-hilo (Windows)

ÍNDICE

Introducción: Proceso vs. Hilo

Antes de empezar, aclaremos la diferencia fundamental:

- Proceso: Es un programa en ejecución (como Excel, Chrome, o tu script de PowerShell). Tiene su propio espacio de memoria aislado, su identificador (PID) y sus propios recursos (como archivos abiertos). Piénsalo como un "contenedor" independiente.

- Hilo (Thread): Es la unidad básica de ejecución dentro de un proceso. Un proceso tiene al menos un hilo (el principal), pero puede crear muchos más. Todos los hilos de un mismo proceso comparten el mismo espacio de memoria y los mismos recursos. Son las "manos" que hacen el trabajo dentro del contenedor.

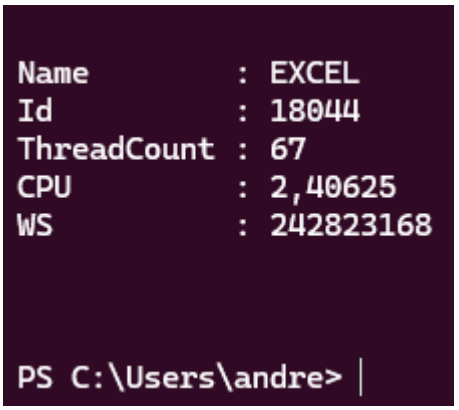
1. Identificación Inicial (Excel)

Vamos a inspeccionar un proceso existente como Excel. Abre un par de hojas de cálculo de Excel para que tenga algo de carga.

1. Abre una ventana de PowerShell.

2. Ejecuta el siguiente comando para encontrar los procesos de Excel y ver sus detalles clave:

```
Get-Process -Name EXCEL | Select-Object Name, Id,
@{Name="ThreadCount"; Expression={$_.Threads.Count}}, CPU, WS
```



Verás que cada proceso de Excel (si tienes varios abiertos) tiene su propio ID, su propia memoria (WS) y su propio conjunto de hilos. Los hilos (ej. 67 en el primer proceso) comparten esos 242MB de memoria para trabajar juntos en esa instancia de Excel.

2. Creación de Carga Controlada

Ahora crearemos dos procesos (dos scripts de PowerShell) que compiten por diferentes recursos.

Scripts de Carga

Script 1: CPU-Intensivo (Script-CPU.ps1) Este script realiza un cálculo matemático en un bucle infinito para saturar un núcleo de CPU.

```
# Script-CPU.ps1
Write-Host "Iniciando bucle de CPU intensivo..."
Write-Host "Presiona Ctrl+C para detener."
$i = [double]1.0
while ($true) {
    # Operación matemática constante para consumir CPU
    $i = [Math]::Tan($i)
}
```

Script 2: I/O-Intensivo (Script-IO.ps1) Este script copia un archivo grande repetidamente para saturar el disco (lectura/escritura).

Antes de ejecutar, crea un archivo "dummy" de 1GB para usarlo como fuente. Abre PowerShell y ejecuta:

fsutil file createnew C:\Temp\dummyfile.dat 1073741824

```
# Script-IO.ps1
Write-Host "Iniciando bucle de I/O (disco) intensivo..."
Write-Host "Presiona Ctrl+C para detener."

$sourceFile = "C:\Temp\dummyfile.dat"
$destinationFolder = "C:\Temp\IO_Test\"

# Asegurarse de que el directorio de destino existe
if (-not (Test-Path $destinationFolder)) {
    New-Item -Path $destinationFolder -ItemType Directory |
Out-Null
}

$i = 0
while ($true) {
    $i++
    $destFile = Join-Path $destinationFolder "copia_$i.dat"
    Write-Host "Copiando a $destFile"
    Copy-Item -Path $sourceFile -Destination $destFile -Force
    Remove-Item -Path $destFile -Force
```

```
}
```

Si observamos el uso

Podemos ver el uso de la CPU y la escritura.

Para obtener el PID:

```
Get-Process -Name powershell
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
852	3117	72976	93260	13,84	2900	1	powershell
685	33	73828	93520	89,59	16088	1	powershell
686	33	61936	78512	0,48	18700	1	powershell

```
$procesoCPU = Get-Process -Id 16088

$procesoIO = Get-Process -Id 2900
```

Para bajar la prioridad usa:

```
$procesoCPU.PriorityClass = "BelowNormal"
```

Para elevar la prioridad usa:

```
$procesoCPU.PriorityClass = "High"
```

Para devolver a su prioridad normal

```
$procesoCPU.PriorityClass = "Normal"
```

3. Trabajo (Job Object)

Aquí hay una confusión común. El comando New-Job / Start-Job de PowerShell crea un "Background Job" de PowerShell, que es una forma de ejecutar scripts de forma asíncrona en segundo plano.

Esto es diferente de un "Job Object" de Windows, que es una característica del sistema operativo (a bajo nivel) para agrupar procesos y aplicarles cuotas estrictas de recursos (CPU, memoria). Esto último no se puede hacer fácilmente con New-Job.

Sin embargo, podemos simular la agrupación que mencionas usando Start-Job.

Usando Start-Job (Agrupación lógica)

```
# Inicia ambos scripts como trabajos en segundo plano
# (Nota: usamos ScriptBlocks aquí en lugar de los archivos)
$jobCPU = Start-Job -ScriptBlock {
    $i = [double]1.0
    while ($true) { $i = [Math]::Tan($i) }
}

$jobIO = Start-Job -ScriptBlock {
    $sourceFile = "C:\Temp\dummyfile.dat"
    $destinationFolder = "C:\Temp\IO_Test\"
    if (-not (Test-Path $destinationFolder)) { New-Item -Path
$destinationFolder -ItemType Directory | Out-Null }
    $i = 0
    while ($true) {
        $i++; $destFile = Join-Path $destinationFolder
"copia_$i.dat";
        Copy-Item -Path $sourceFile -Destination $destFile -Force;
        Remove-Item -Path $destFile -Force
    }
}
```

Ahora puedes gestionarlos como un grupo:

```
# Ver todos los trabajos que se están ejecutando en esta sesión
Get-Job

# Detener TODOS los trabajos
Get-Job | Stop-Job

# Limpiar (eliminar de la lista)
```

```
Get-Job | Remove-Job
```

Ventajas de Gestionar Procesos como un "Trabajo"

Aunque Start-Job de PowerShell es para concurrencia, las ventajas que mencionas (control centralizado) se refieren al concepto de Job Object de Windows (usado por servidores, Docker, etc.):

- Control de Ciclo de Vida: Es la ventaja principal. Puedes agrupar múltiples procesos (ej. un servidor web y sus procesos "worker"). Si "matas" el Job Object, Windows garantiza que todos los procesos dentro de él (y cualquier proceso que ellos inicien) mueran instantáneamente. Esto es crucial en servidores para evitar "procesos huérfanos" que consumen recursos.

- Límites de Recursos (Quotas): Esto es lo que no hace Start-Job. Un Job Object real de Windows puede imponer límites estrictos:

    - "No permitas que este grupo de procesos use más del 20% de la CPU total".
    - "Limita a este grupo a un máximo de 500MB de RAM".
    - "Asigna este grupo solo a los núcleos de CPU 4, 5 y 6" (Afinidad).

- Aislamiento y Seguridad: Se pueden usar para restringir lo que el grupo de procesos puede hacer (ej. impedir que accedan a la red o al portapapeles).

En resumen, los Jobs de PowerShell son para multitarea en tus scripts. Los Job Objects de Windows son para administración y aislamiento de recursos a nivel de sistema operativo.

4. Registro y Monitorización

Para esta parte, Process Explorer (de la suite Sysinternals de Microsoft) es superior al Administrador de Tareas, ya que te permite ver los hilos individuales de un proceso. El Monitor de Recursos (resmon.exe) es el mejor para ver I/O de disco.

Crear un Log con PowerShell:

Puedes ejecutar este script en una tercera terminal (como administrador) mientras tus scripts de carga están activos y mientras cambias las prioridades.

```
# Script-Logger.ps1
$cpuPID = <PID_CPU>  # Reemplaza con el ID de tu Script-CPU
$ioPID = <PID_IO>    # Reemplaza con el ID de tu Script-IO

$logEntries = @()
Write-Host "Iniciando monitorización... Presiona Ctrl+C para
detener y guardar."

try {
    while ($true) {
        $pCPU = Get-Process -Id $cpuPID
        $pIO = Get-Process -Id $ioPID

        $snapshot = [PSCustomObject]@{
            Timestamp    = Get-Date -Format "HH:mm:ss"
            CPU_Priority = $pCPU.PriorityClass
            CPU_Threads  = $pCPU.Threads.Count
            CPU_Usage_s  = $pCPU.CPU # Nota: Este es el total
acumulado
            CPU_Memory_MB= [Math]::Round($pCPU.WS / 1MB)
            IO_Priority  = $pIO.PriorityClass
            IO_Threads   = $pIO.Threads.Count
            IO_Usage_s   = $pIO.CPU
            IO_Memory_MB = [Math]::Round($pIO.WS / 1MB)
        }

        $logEntries += $snapshot
        $snapshot # Muestra el snapshot en la consola

        Start-Sleep -Seconds 3
    }
```

```
}
catch {
    Write-Warning "Proceso no encontrado o detenido."
}
finally {
    # Al presionar Ctrl+C, guardará el log antes de salir
    $logEntries | Export-Csv -Path
"C:\Temp\process_monitor_log.csv" -NoTypeInformation -Encoding
UTF8
    Write-Host "Log guardado en C:\Temp\process_monitor_log.csv"
}
```

Ahora, ejecuta tus scripts de carga, ejecuta el logger, y luego cambia las prioridades (Paso 2). Cuando termines, detén el logger (Ctrl+C) y tendrás un archivo .csv en C:\Temp que muestra cómo cambiaron los valores (especialmente CPU_Priority) durante el experimento.

Conclusión

Tras realizar este experimento, estas son las conclusiones clave:

Diferencias Observadas

- **Procesos (Tus dos scripts `powershell.exe`):**
    - **Consumo:** Cada uno tenía su propio contador de CPU y su propia asignación de memoria (WS).
    - **Aislamiento:** Eran completamente independientes. Si uno crasheaba, el otro seguía. El planificador de Windows los trataba como dos "clientes" separados pidiendo recursos (uno pidiendo CPU, el otro pidiendo Disco).
    - **Coste:** Iniciar cada `powershell.exe` fue una operación "pesada" para el sistema (cargó el intérprete, asignó memoria, etc.).
- **Hilos (Los hilos dentro de `EXCEL.EXE`):**
    - **Consumo:** Todos los hilos (ej. 45) *compartían* la misma memoria del proceso Excel. El uso de CPU del proceso era la *suma* del trabajo de todos sus hilos.
    - **Aislamiento:** No hay. Si un hilo (ej. el del corrector ortográfico) tiene un error grave y corrompe la memoria, todo el proceso `EXCEL.EXE` crashea.
    - **Coste:** Crear un nuevo hilo es extremadamente "barato" y rápido en comparación con crear un proceso.

¿Cuándo usar Hilos vs. Procesos?

Usa Múltiples Hilos (Multithreading) si:

- Necesitas realizar tareas **paralelas *dentro* de una misma aplicación** que deben **compartir datos** fácilmente.
- Quieres mantener tu aplicación "fluida". (Ej. un hilo para la interfaz de usuario, otro para guardar en segundo plano).

- **Ejemplo:** Excel usa un hilo para que puedas escribir (UI), otro para recalcular fórmulas y otro para el autoguardado. Todos trabajan sobre el *mismo* documento en memoria.

Usa Múltiples Procesos (Multiprocessing) si:

- Necesitas **aislamiento y estabilidad**. Si una tarea puede crashear, aislarla en su propio proceso evita que tumbe el resto del sistema.
- Necesitas **seguridad**. Puedes restringir un proceso (ej. un "sandbox").
- Las tareas son **completamente independientes** y no necesitan compartir memoria.
- **Ejemplo:** Google Chrome crea un proceso separado para cada pestaña. Si una pestaña (un proceso) crashea, solo se cierra esa pestaña, no todo el navegador.

Estrategia de Planificación y Equilibrio

1. **Identificar el Cuello de Botella:** El primer paso es siempre usar el **Monitor de Recursos**. ¿Qué recurso está al 100%? ¿CPU, Disco, Red o Memoria? No tiene sentido optimizar la CPU si el problema es el disco.
2. **Usar Prioridades (Priorities):** Es la herramienta principal para equilibrar la carga.
    - **Tareas de Fondo (Batch):** Tareas pesadas que no son urgentes (copias de seguridad, renderizado de vídeo, análisis de datos) deben ejecutarse con prioridad `BelowNormal` (Baja). El sistema las ejecutará "en su tiempo libre".
    - **Tareas Interactivas (UI):** Aplicaciones que el usuario está usando activamente (Word, navegador) deben estar en `Normal`.
    - **Tareas Críticas:** Procesos del sistema (drivers de audio, ratón) se ejecutan en `High` para garantizar que la respuesta sea instantánea. (Casi nunca debes usar `RealTime`).
3. **Usar Afinidad (Affinity):** En servidores, si tienes un proceso de base de datos muy intensivo, puedes "atarlo" (establecer afinidad) a núcleos de CPU específicos (ej. núcleos 4-7), dejando los núcleos 0-3 libres para el sistema operativo y otras aplicaciones, garantizando que el servidor siempre responda.