2.3- OPTIMIZACIÓN DE PROCESOS MULTI-HILO (WINDOWS)

Cristóbal Suárez Abad

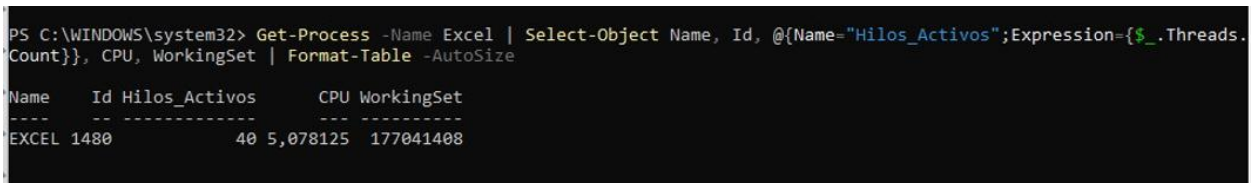
ADMINISTRACIÓN DE SISTEMAS OPERATIVOS - 2º ASIR

Contenido

Identificación inicial:

- *Utiliza **PowerShell** para identificar los procesos que ejecuta la aplicación de ofimática (Excel) y sus hilos asociados. Documenta cuántos hilos están activos por cada proceso y qué consumo de CPU y memoria tienen. (Get-Process | Select Name,Id,Threads)*

Get-Process -Name Excel | Select-Object Name, Id, @{Name="Hilos_Activos";Expression={$_.Threads.Count}}, CPU, WorkingSet | Format-Table -AutoSize



Indica el ID del proceso, los hilos que está usando el tiempo acumulado de uso de la CPU (5 segundos y pico) y la RAM en bytes (unos 168MB).

Creación de carga controlada:

- *Crea dos scripts del powerShell para ejecutar dos procesos intensivos al mismo tiempo:*
    - *Uno con cálculo (por ejemplo, un script de PowerShell que calcule el factorial de un numero).*
    - *Otro con lectura/escritura en disco (por ejemplo, copiar varios archivos grandes, una copia de seguridad).*
- *Observa cómo Windows distribuye los recursos entre ambos.*
- *Ve cambiando las prioridades de los proceso para ver como afecta al rendimiento del sistema.*

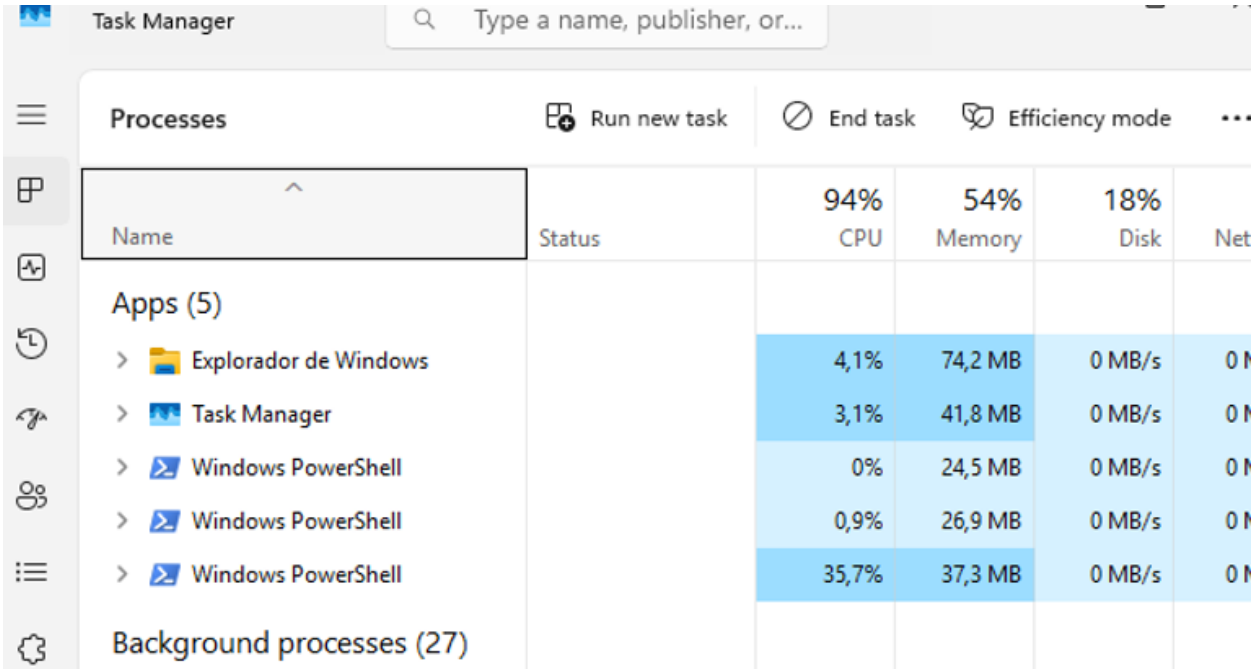
- Scritp de cálculo factorial:

```
function Get-Factorial($n) {
  $result = [System.Numerics.BigInteger] 1
  for ($i = 2; $i -le $n; $i++) {
    $result = $result * $i
  }
  return $result
}

$start_time = Get-Date
Write-Host "Iniciando cálculo intensivo..."
# Aumentamos el número a un valor que debería forzar un tiempo de ejecución mayor
# (El tiempo exacto variará por sistema)
Get-Factorial 80000 | Out-Null
$end_time = Get-Date
$duration = $end_time - $start_time
Write-Host "Cálculo completado en: $($duration.TotalSeconds) segundos."
```

- Script de lectura y escritura (I/O):

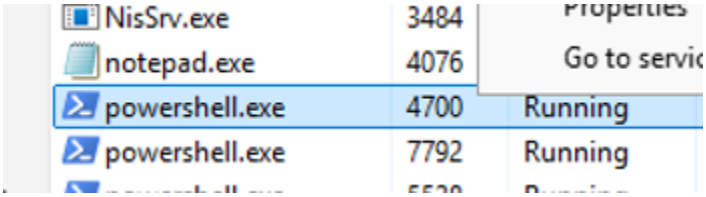
```
$sourceFile = "Z:\debian-13.0.0-amd64-netinst.iso" # ¡Asegúrate de que este archivo exista!
$destinationDir = "Z:\prueba_proceso" # ¡Asegúrate de que esta carpeta exista!
$copyCount = 3 # Número de copias

Write-Host "Iniciando operación I/O intensiva..."

Measure-Command {
  for ($i = 1; $i -le $copyCount; $i++) {
    $destinationPath = Join-Path -Path $destinationDir -ChildPath "Copia_$i_$(Get-Date -Format 'HHmmss').dat"
    Copy-Item -Path $sourceFile -Destination $destinationPath -Force
    Write-Host "Copia $i completada."
  }
} | ForEach-Object {
  Write-Host "Operación de copia completada en: $($_.TotalSeconds) segundos."
}
```

Los ejecutamos y vemos como el proceso de calculo factorial consumo gran parte de la CPU:



Podemos identificar sus ID con:
Get-Process -Name powershell

O mirando en el Administrador de Tareas:



Para bajar la prioridad al proceso usamos:
(Get-Process -Id 4700).PriorityClass = "Idle"

Ese es el más bajo, pero estos son todos los posibles:

(Get-Process -Id 4700).PriorityClass = "Idle" # prioridad mínima
(Get-Process -Id 4700).PriorityClass = "BelowNormal" # ligeramente baja
(Get-Process -Id 4700).PriorityClass = "Normal"
(Get-Process -Id 4700).PriorityClass = "AboveNormal"
(Get-Process -Id 4700).PriorityClass = "High"

Trabajo (Job Object):

- *Agrupa varios procesos bajo un mismo "Job" utilizando PowerShell (New-Job).*
- *Limita su uso de CPU o memoria y analiza el resultado.*
- *Explica qué ventajas tiene gestionar un grupo de procesos como un único trabajo.*

(El control centralizado sobre grupos de procesos relacionados es ideal para servidores o entornos multiusuario).

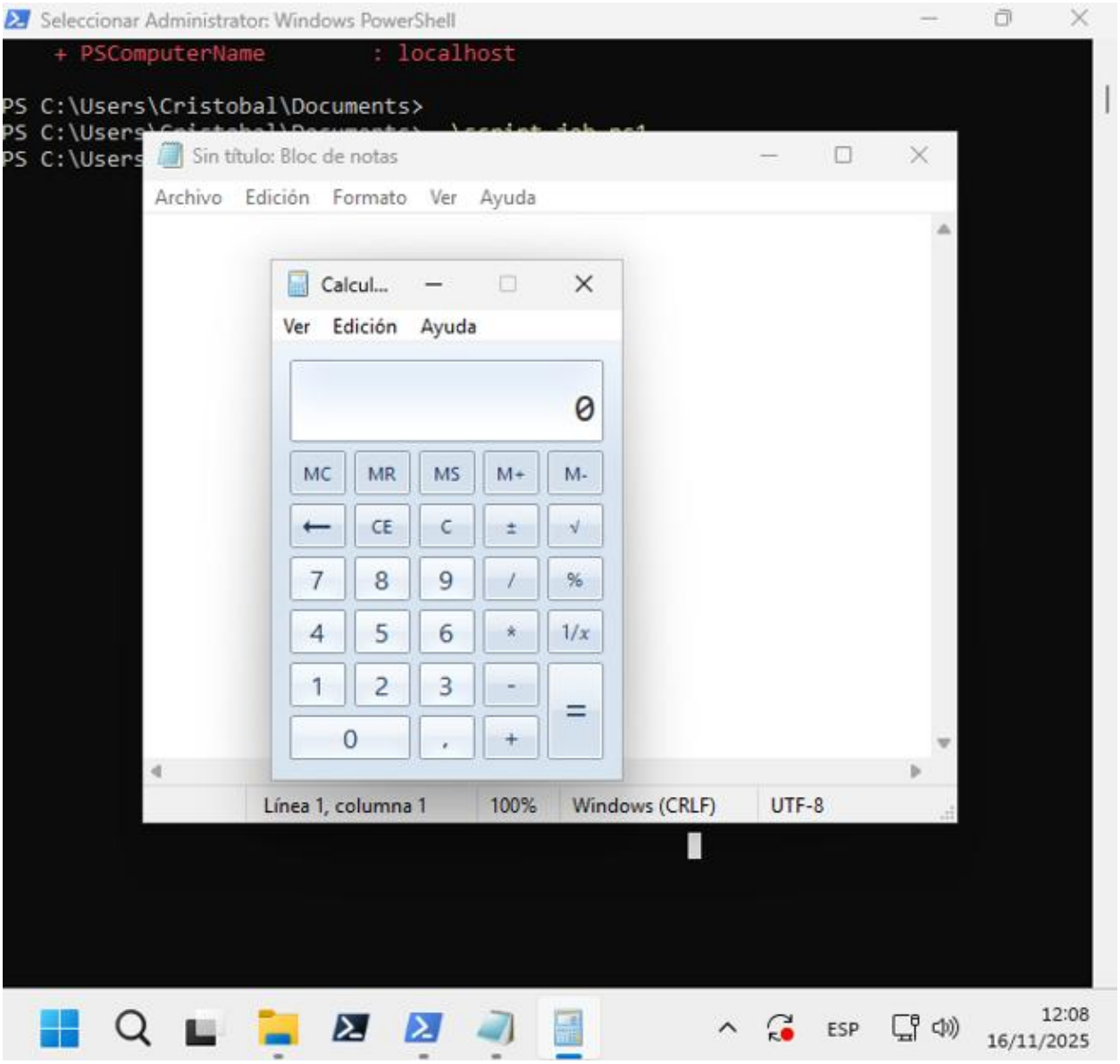
Vamos a crear un Job que agrupe varios procesos. Por ejemplo, lanzamos notepad, calc y mspaint bajo un mismo Job:

```
$job = Start-Job -ScriptBlock {
  # Guardamos los procesos en variables para control
  $proc1 = Start-Process notepad -PassThru
  $proc2 = Start-Process calc -PassThru
  $proc3 = Start-Process mspaint -PassThru

  # Mantener el Job activo para poder monitorizarlo
  Start-Sleep -Seconds 60
}
```

-PassThru devuelve el objeto del proceso para poder acceder a sus propiedades.

Start-Sleep mantiene el Job activo para poder monitorear recursos.

Cuando lo ejecutemos se abrirán los programas indicados en el script.



Ahora vamos a hacer uno para limitar. PowerShell no puede limitar directamente la CPU de un proceso, pero podemos usar Job Objects de Windows o asignar afinidad de CPU y prioridad:

```
$jobProcesses = Get-Process | Where-Object { $_.ProcessName -in "notepad","calc","mspaint" }

foreach ($p in $jobProcesses) {
  # Limitar afinidad a 1 CPU
  $p.ProcessorAffinity = 1

  # Cambiar prioridad (Reduce consumo de CPU)
  $p.PriorityClass = "BelowNormal"
}
```

ProcessorAffinity limita en qué núcleo de CPU puede ejecutarse.

PriorityClass reduce el impacto sobre el resto del sistema.

```
PS C:\Users\Cristobal\Documents> Get-Process -Name "notepad" | Select-Object -Prop
erty Name, Id, PriorityClass

Name      Id PriorityClass
----      -- -------------
notepad 2612   BelowNormal


PS C:\Users\Cristobal\Documents> Get-Process -Name "mspaint" | Select-Object -Prop
erty Name, Id, PriorityClass

Name      Id PriorityClass
----      -- -------------
mspaint 7032   BelowNormal


PS C:\Users\Cristobal\Documents> _
```

Para memoria, se puede monitorizar y alertar si excede cierto valor, aunque PowerShell no restringe memoria directamente.

Ver el estado del Job
Get-Job

Recibir salida del Job (si hay)
Receive-Job -Id $job.Id

Analizar uso de CPU y memoria de los procesos del Job
$jobProcesses | Select-Object Name, Id, CPU, WS

Registro y monitorización:

- *Activa el Monitor de recursos y Process Explorer para registrar los valores de CPU, memoria e hilos de ambos procesos.*
- *Guarda un log con la evolución de cada proceso (inicio, consumo, prioridad, fin). Este archivo te permitirá comprobar los consumos antes y después de cambiar las prioridades.*

El Monitor de recursos ya viene instalado: resmon.exe

Pero **Process Explorer** debemos descargarlo: https://learn.microsoft.com/es-es/sysinternals/downloads/process-explorer

Ahora vamos a crear un log que guarde el estado de los procesos que hemos creado previamente (el de cálculo factorial y el de Lectura/Escritura). Este script registra cada segundo, por 30 segundos, la actividad de un proceso (hay que indicar su ID).

```
# ID del proceso que deseas monitorear
$ID_del_Proceso = 5064 # ¡Reemplaza <ID_del_Proceso> con el ID real!

# Ruta del archivo de log
$Ruta_Log = '.\Log_Procesos.csv'

# Número de veces que se ejecutará el bucle (30 segundos / 1 segundo de espera = 30 veces)
$Contador_Maximo = 30
$Contador = 0

Write-Host "Iniciando el monitoreo del Proceso con ID: $ID_del_Proceso. Registrando cada segundo por 30 segundos..."

# Bucle que se ejecuta 30 veces
while ($Contador -lt $Contador_Maximo) {
  try {
    # 1. Obtener los datos del proceso
    $Proceso = Get-Process -Id $ID_del_Proceso -ErrorAction Stop

    # 2. Estructura de los datos de log
    $Datos = [PSCustomObject]@{
      TimeStamp = Get-Date -Format 'yyyy-MM-dd HH:mm:ss'
      Nombre = $Proceso.Name
      ID = $Proceso.Id
```

```
        Prioridad = $Proceso.PriorityClass
        CPU_Segundos = $Proceso.CPU
        Memoria_WS_MB = ($Proceso.WS / 1MB).ToString("N2")
        Hilos = $Proceso.Threads.Count
        Estado = "Ejecucion" # Se registra cada segundo mientras está en ejecución
      }

      # 3. Exportar los datos al CSV
      # El encabezado se añade solo la primera vez ($Contador -eq 0)
      if ($Contador -eq 0 -and -not (Test-Path $Ruta_Log)) {
        $Datos | Export-Csv -Path $Ruta_Log -NoTypeInformation -Force
      } else {
        $Datos | Export-Csv -Path $Ruta_Log -Append -NoTypeInformation
      }

      Write-Host "Registro #$($Contador + 1) añadido correctamente."

    } catch {
      # **CORRECCIÓN APLICADA AQUÍ:** Uso de $($ID_del_Proceso) para evitar el
ParserError
      Write-Warning "ERROR al obtener el proceso con ID $($ID_del_Proceso):
$($_.Exception.Message)"
      # Si falla, salimos del bucle
      break
    }

    # 4. Aumentar el contador
    $Contador++

    # 5. Esperar 1 segundo antes de la próxima iteración (excepto en la última)
    if ($Contador -lt $Contador_Maximo) {
      Start-Sleep -Seconds 1
    }
}

Write-Host "Monitoreo de 30 segundos finalizado. Revise el archivo $Ruta_Log"
```

Conclusión:

- *Explica las diferencias observadas entre procesos e hilos en términos de consumo y eficiencia.*
- *Evalúa cuándo conviene usar varios hilos frente a varios procesos.*
- *Propón una estrategia de planificación para equilibrar carga y rendimiento.*

Los **hilos son más eficientes** para el trabajo cooperativo dentro de una misma aplicación debido a su bajo coste de creación y la velocidad de comunicación (no tienen que "copiar" o "serializar" datos). Los **procesos son más seguros y estables** para tareas que deben estar aisladas.

Característica	**Proceso**	**Hilo (Thread)**
Recursos	Entidad independiente, con su propio espacio de **memoria virtual** y descriptores de recursos.	Entidad de ejecución dentro de un proceso, **comparte** la memoria y recursos del proceso padre.
Creación	Pesada (alto *overhead*). Requiere que el SO asigne y configure nuevos espacios de memoria.	Ligera (bajo *overhead*). Solo necesita un nuevo *Stack* y *Thread Control Block*.
Comunicación	Lenta. Requiere comunicación entre procesos (IPC) como *pipes* o *shared memory*.	Rápida. Acceso directo y compartido a la misma memoria del proceso.
Fallo	Un fallo de un proceso **no afecta** a otros procesos.	Un fallo de un hilo **puede causar el fallo** de todo el proceso.

Evaluación: Hilos vs. Procesos

Situación	Recomendación	Razón
Tareas de Cálculo Intensivo (CPU-Bound)	**Hilos (Multithreading)**	Los hilos permiten usar múltiples núcleos de CPU para resolver una parte del problema en paralelo, compartiendo los datos de entrada en memoria (ej. procesamiento de imágenes, cálculos científicos).
Tareas de E/S Intensiva (I/O-Bound)	**Hilos o Procesos Ligeros**	Mientras un hilo/proceso espera una operación de disco o red, otro puede ejecutar código. Los hilos son más eficientes por el bajo *overhead* (ej. un servidor web).
Tareas Discretas e Independientes	**Procesos (Multiprocessing)**	Si las tareas están totalmente separadas y necesitan aislamiento y tolerancia a fallos (ej. Navegador web - cada pestaña es un proceso), los procesos son mejores.

Estrategia de Planificación para Equilibrio de Carga y Rendimiento

La estrategia óptima se basa en la priorización y el aislamiento:

1. **Priorización Basada en la Necesidad:**
    - **Prioridad Alta:** Procesos que manejan la **interfaz de usuario (UI)** y la **latencia** (ej. *streaming*, tiempo real, controladores) para asegurar una experiencia de usuario fluida.
    - **Prioridad Normal:** La mayoría de las aplicaciones de usuario y de servidor.
    - **Prioridad Baja (BelowNormal/Idle):** Tareas de **fondo no urgentes** (ej. *backups*, indexación, compilaciones en segundo plano). Esto permite que el trabajo se complete sin comprometer la capacidad de respuesta del sistema.
2. **Uso de Job Objects para Aislamiento (Servidores/Multiusuario):**
    - **Aislamiento de Recursos:** Encapsular servicios no críticos (ej. un servicio de análisis) en un Job Object con un límite estricto de CPU y memoria (ej. 20% de CPU). Esto garantiza que, incluso si el servicio falla o tiene un consumo descontrolado, **nunca comprometerá los recursos del SO o de servicios críticos**.
3. **Diferenciación CPU vs. I/O:**
    - El planificador de Windows tiende a favorecer los procesos de I/O sobre los de CPU, ya que las tareas de I/O a menudo esperan la finalización de operaciones lentas.
    - **Estrategia:** No elevar la prioridad de las tareas intensivas en I/O a menos que sea crítico, ya que saturarán el bus de datos. Aplicar **prioridad baja** a las tareas de **cálculo intensivo no urgente** asegura que el SO siempre tenga núcleos de CPU disponibles para tareas que requieren una respuesta rápida, logrando el equilibrio entre rendimiento y *throughput* (cantidad de trabajo completado).