Un freeze total del sistema (mouse congelado, audio en bucle, apagado forzado) en Windows con PyTorch es síntoma casi inequívoco de un conflicto a muy bajo nivel. Dado que ya has descartado el "OOM" clásico (que simplemente mataría el proceso Python) y el sobrecalentamiento, aquí está mi diagnóstico profundo.

El problema reside casi con total seguridad en la intersección de Multithreading (Python) + OpenMP (PyTorch CPU backend) + Windows.

1. El "Smoking Gun": Threading vs OpenMP

El sospechoso principal está en `manager.py`.

Python

```
# manager.py
def launch_job(self, job: Job):
    # ...
    thread = threading.Thread(target=self._run_job, args=(job.id,), daemon=True)
    thread.start()
```

El Conflicto:

Estás lanzando la inferencia de PyTorch dentro de un threading.Thread secundario.

PyTorch, cuando corre en CPU (o hace pre/post-proceso en CPU antes de enviar a GPU), utiliza internamente OpenMP para paralelizar operaciones matemáticas.

En Windows, inicializar OpenMP dentro de un hilo hijo de Python (que no es el hilo principal) mientras el hilo principal está activo (uvicorn/FastAPI loop) provoca frecuentemente Deadlocks o Race Conditions en el runtime de C++ (libiomp5md.dll o mkl).

Esto hace que todos los núcleos de la CPU entren en un estado de espera activa infinita a nivel de kernel, lo que congela el sistema operativo completo, no solo Python.

2. Análisis del Código y Puntos Críticos

A. La expansión de máscaras (`sam3_runner.py`)

Aunque el threading es la causa raíz más probable, el detonante del consumo de recursos es este bloque:

Python

```
# sam3_runner.py
target_sizes = [(orig_height, orig_width)] # <--- ¡PELIGRO!
# ...
results = self.processor.post_process_instance_segmentation(
    outputs,
    threshold=0.0,
    mask_threshold=self.mask_threshold,
    target_sizes=target_sizes, # <--- Aquí
)[0]
```

- El problema: Incluso si `target_long_side` es 512 (bajita resolución para inferencia), le estás pidiendo al `processor` que reescale las máscaras resultantes al tamaño original de la imagen (`orig_height, orig_width`).
- Escenario: Si tu imagen original es de 4000x3000 píxeles (12MP) y SAM detecta 50 objetos, `post_process` intentará crear 50 mapas de bits de 12MP en RAM simultáneamente para interpolarlos.

- Resultado: En una máquina con 16GB RAM y CPU antigua (i7-4ª Gen), esto puede saturar el bus de memoria instantáneamente, provocando que Windows intente paginar a disco de golpe, congelando la UI.

B. `gc.collect()` dentro del bucle

En `manager.py`:

Python

```
# manager.py
try:
    pil_img = load_image(Path(img.abs_path))
# ...
```

No veo llamadas explícitas a `gc.collect()` por imagen en `manager.py`, solo en `sam3_runner.py` después de un error o reintento.

- Recomendación: Python retiene memoria agresivamente. En sistemas con RAM justa (16GB compartidos con OS y GPU), procesar imágenes en bucle sin un `gc.collect()` explícito al final de cada iteración exitosa puede fragmentar la RAM hasta que el sistema colapsa.

3. Experimentos de Aislamiento

Para confirmar la causa, te propongo realizar estos 3 experimentos en orden (de menor a mayor intrusión).

Experimento A: Anular Multithreading (La prueba de fuego)

Elimina la variable de la ecuación del hilo secundario. Modifica `manager.py` para correr el job en el hilo principal (bloqueará la API, pero nos dirá si es el threading).

Cambio en `manager.py`:

Python

```
def launch_job(self, job: Job):
    # COMENTAR ESTO:
    # cancel_event = threading.Event()
    # self.active_jobs[job.id] = cancel_event
    # thread = threading.Thread(target=self._run_job, args=(job.id,),
daemon=True)
    # thread.start()

    # AÑADIR ESTO (Ejecución síncrona):
    self.active_jobs[job.id] = threading.Event()
    self._run_job(job.id)
```

- Hipótesis: Si esto no congela el PC, el problema es 100% el conflicto Threading/OpenMP.

Experimento B: Restringir OpenMP

Si necesitas mantener el threading, debes castrar la capacidad de PyTorch de usar múltiples hilos de CPU internamente.

Añade esto al inicio absoluto de manager.py (antes de importar torch/transformers):

Python

```
import os
os.environ["OMP_NUM_THREADS"] = "1"
os.environ["MKL_NUM_THREADS"] = "1"
os.environ["PYTORCH_NUM_THREADS"] = "1"
```

```
# from ... import ...
```

- Hipótesis: Esto evitará que los hilos nativos de PyTorch peleen con el Global Interpreter Lock (GIL) y el thread de Python.

Experimento C: Eliminar Post-procesado Masivo

Si A y B fallan, el problema es el coste de memoria de las máscaras. Modifica `sam3_runner.py` para no escalar las máscaras al tamaño original.

Cambio en `sam3_runner.py`:

Python

```
# En run_pcs:
# target_sizes = [(orig_height, orig_width)]  <-- Comentar esto
target_sizes = None # Dejar que devuelva el tamaño nativo del modelo (ej. 256x256)
```

- Esto te dará máscaras pequeñas (que luego tendrás que escalar tú si las necesitas), pero aliviará la CPU/RAM dramáticamente.

4. Diagnóstico final y Antipatrones Windows

1. Deadlock de Librerías (Alta Probabilidad): Al usar `uvicorn` (asíncrono) + `threading` (hilo estándar) + `PyTorch` (OpenMP) en Windows, se crea una tormenta perfecta. Las librerías de Intel (MKL) que PyTorch usa por debajo a veces no manejan bien el fork/spawn de hilos en Windows si no se configuran.
2. Antipatrón detectado: No uses `daemon=True` para hilos que hacen computación pesada (NumPy/PyTorch). Si el programa principal intenta cerrar o interactuar, los hilos daemon se mueren abruptamente, pudiendo dejar la GPU o drivers en estados inconsistentes.
3. Hardware (i7-4790S): Tu CPU es Haswell. Soporta AVX2, pero PyTorch moderno está optimizado para AVX-512 o instrucciones más nuevas. A veces, la emulación o el "fallback" de instrucciones en CPUs antiguas bajo carga pesada en Windows puede causar inestabilidad (menos probable, pero posible).

Próximo paso

Te sugiero encarecidamente que empieces por el Experimento A (Ejecución Síncrona). Es el más revelador.

¿Quieres que te genere una versión modificada de `manager.py` que implemente la ejecución síncrona y la limpieza de memoria forzada por iteración para probarlo inmediatamente?