



Universidad Nacional Autónoma de México

Tarea 01 - Ejercicios de sincronización

Curso: Sistemas Operativos

Profesor: Dr. Gunnar Eyal Wolf Iszaveich

Autor: Chávez López B. Alejandro

Cuenta: 421112601

Grupo: 08

Proyecto de Sincronización – Problema del Elevador

Planteamiento

El elevador de la Facultad se descompone demasiado, porque sus usuarios no respetan los límites. Te toca evitar este desgaste (y el peligro que conlleva). Implementa el elevador como un hilo, y a cada persona que quiere usarlo como otro hilo. El elevador de la Facultad de Ingeniería da servicio a cinco pisos. Un usuario puede llamarlo en cualquiera de ellos. Puede querer ir a cualquiera otro de ellos.

Reglas

- El elevador tiene capacidad para cinco pasajeros.
- Para ir del piso x a y , el elevador tiene que cruzar todos los pisos intermedios.
- Los usuarios prefieren esperar dentro del elevador que fuera de él.
- Si el elevador va subiendo y pasa por el piso x , donde está A esperando para bajar, A aborda al elevador (no espera a que vaya en la dirección correcta)..

Lenguaje y entorno de desarrollo

El programa fue desarrollado en **Python 3.3.13.7**, utilizando el módulo **threading**. El programa fue probado en **Linux (Fedora 42)**.

Para ejecutarlo:

```
1 python3 ejerciciosSincElevador.py
```

(No requiere dependencias adicionales.)

Estrategia de sincronización

Semáforo de capacidad

Se implementó un semáforo con valor máximo igual a la capacidad del elevador (5 usuarios). Este semáforo controla que no ingresen más usuarios de los permitidos simultáneamente:

```
1 capacidad = threading.Semaphore(CAPACIDAD_MAX)
2 ...
3 capacidad.acquire()
4 print(f"Usuario {id} abordo en piso {origen}")
5 ...
6 capacidad.release()
```

Cada usuario que aborda el elevador ejecuta `acquire()` para ocupar un espacio. Cuando el usuario llega a su destino y desciende, llama a `release()` para liberar el lugar. Así, nunca habrá más de cinco usuarios dentro del elevador.

Exclusión mutua con Lock

Las variables compartidas `piso_actual`, `direccion` y la lista `usuarios_en_elevador` son modificadas tanto por el hilo del elevador como por los hilos de los usuarios. Para evitar que hubieran cambios al mismo tiempo que pudieran generar inconsistencias, se utilizó un candado (`Lock`):

```
1 mutex_estado = threading.Lock()
2 ...
3 with mutex_estado:
4     piso_actual += direccion
5     if piso_actual == NUM_PISOS - 1:
6         direccion = -1
7     elif piso_actual == 0:
8         direccion = 1
```

Mientras un hilo posee este candado, ningún otro puede modificar las variables protegidas.

Variables de condición (Condition)

Se usó `Condition` para que los usuarios esperen a que el elevador llegue a su piso, sin necesidad de verificar constantemente.

```
1 cond_pisos = [threading.Condition() for _ in range(NUM_PISOS)]
2 ...
3 with cond_pisos[piso_actual]:
4     cond_pisos[piso_actual].notify_all()
```

Cuando el elevador llega a un piso, se ejecuta `notify_all()` para despertar a los hilos (usuarios, en este caso) que estaban esperando ahí. Además, cada usuario permanece en espera con:

```
1 with cond_pisos[origen]:
2     cond_pisos[origen].wait_for(lambda: piso_actual == origen)
```

Así hay una comunicación sincronizada entre elevador y usuarios.

Eventos para control de turnos

Ya en la etapa de refinamiento, se añadieron dos eventos para dividir los pisos en dos grupos (bajos y altos) y alternar su atención, evitando que un grupo monopolizara el elevador (ya que en ocasiones se quedaban esperando mucho tiempo algunos usuarios). Cada grupo se activa o desactiva según la posición del elevador en un determinado momento:

```
1 turno_bajo = threading.Event()
2 turno_alto = threading.Event()
3 turno_bajo.set() #Inicia por los pisos bajos
4
5 if piso_actual == 2 and direccion == 1:
6     turno_bajo.clear(); turno_alto.set()
```

```
7 elif piso_actual == 3 and direccion == -1:  
8     turno_alto.clear(); turno_bajo.set()
```

Los usuarios pertenecientes a cada grupo esperan hasta que su turno se active:

```
1 grupo = 'bajo' if origen <= 2 else 'alto'  
2 (turno_bajo if grupo == 'bajo' else turno_alto).wait()
```

Así, el elevador puede atender a todos los pisos de modo más "justo" y evita la inanición.

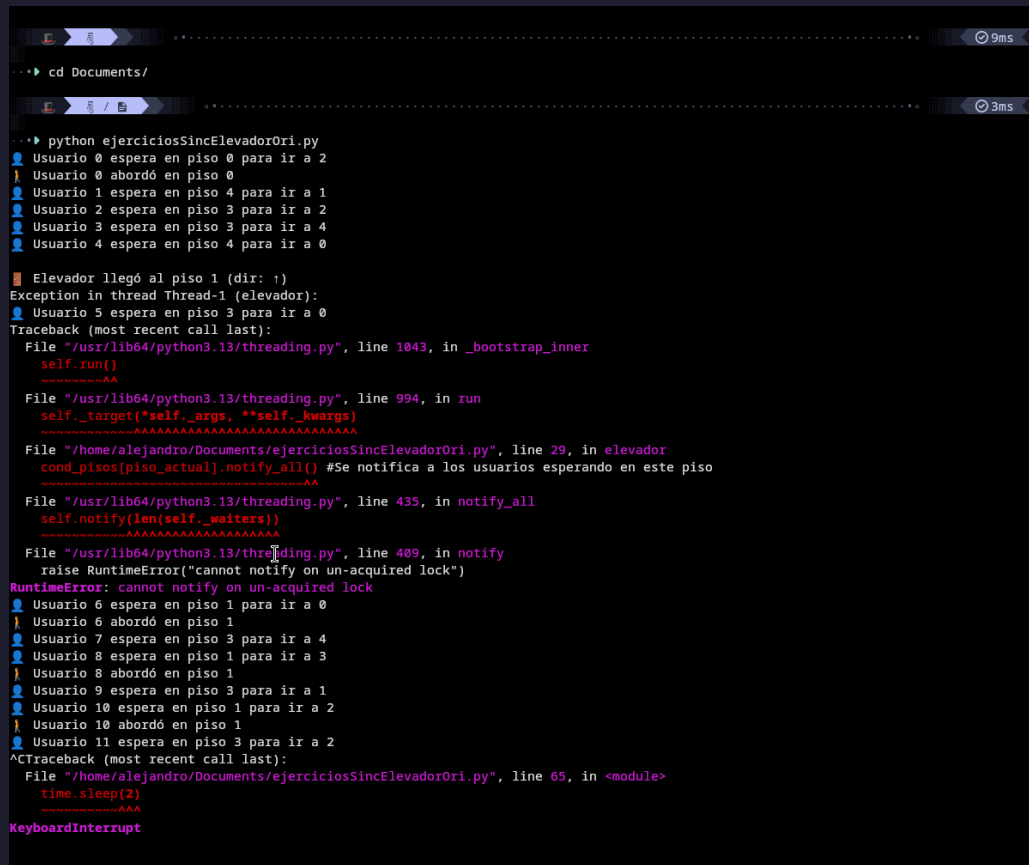
Versión inicial

En la primera versión del programa se corrían bien los hilos de usuarios y del elevador, pero se mostraba la excepción:

```
1 RuntimeError: cannot notify on un-acquired lock
```

cuando se intentaba notificar a los usuarios que esperaban en su piso.

Esto hacía que el hilo del elevador se detuviera, los usuarios esperaran indefinidamente y los que habían logrado abordar ya no se movían.



```
cd Documents/

python ejerciciosSincElevadorOri.py
Usuario 0 espera en piso 0 para ir a 2
| Usuario 0 abordó en piso 0
| Usuario 1 espera en piso 4 para ir a 1
| Usuario 2 espera en piso 3 para ir a 2
| Usuario 3 espera en piso 3 para ir a 4
| Usuario 4 espera en piso 4 para ir a 0

Elevador llegó al piso 1 (dir: ↑)
Exception in thread Thread-1 (elevador):
| Usuario 5 espera en piso 3 para ir a 0
Traceback (most recent call last):
  File "/usr/lib64/python3.13/threading.py", line 1043, in _bootstrap_inner
    self.run()
    ~~~~~^
  File "/usr/lib64/python3.13/threading.py", line 994, in run
    self._target(*self._args, **self._kwargs)
    ~~~~~^
  File "/home/alejandro/Documents/ejerciciosSincElevadorOri.py", line 29, in elevador
    cond_pisos[piso_actual].notify_all() #Se notifica a los usuarios esperando en este piso
    ~~~~~^
  File "/usr/lib64/python3.13/threading.py", line 435, in notify_all
    self.notify(len(self._waiters))
    ~~~~~^
  File "/usr/lib64/python3.13/threading.py", line 409, in notify
    raise RuntimeError("cannot notify on un-acquired lock")
RuntimeError: cannot notify on un-acquired lock
| Usuario 6 espera en piso 1 para ir a 0
| Usuario 6 abordó en piso 1
| Usuario 7 espera en piso 3 para ir a 4
| Usuario 8 espera en piso 1 para ir a 3
| Usuario 8 abordó en piso 1
| Usuario 9 espera en piso 3 para ir a 1
| Usuario 10 espera en piso 1 para ir a 2
| Usuario 10 abordó en piso 1
| Usuario 11 espera en piso 3 para ir a 2
^CTraceback (most recent call last):
  File "/home/alejandro/Documents/ejerciciosSincElevadorOri.py", line 65, in <module>
    time.sleep(2)
    ~~~~~^
KeyboardInterrupt
```

Figure 1: Ejecución (con errores) de la primera versión.

El error indicaba que se estaba llamando a `notify_all()` sin haber adquirido el lock interno del objeto `Condition`. Después de investigar me enteré que en Python, las operaciones `wait()` y `notify_all()` deben ejecutarse dentro del contexto de `Condition`, viéndose así:

```
1 with cond_pisos[piso_actual]:
2     cond_pisos[piso_actual].notify_all()
```

Segunda versión

El programa funcionaba correctamente:

- Los usuarios esperaban en sus pisos.
- El elevador se desplazaba entre los cinco niveles.
- La capacidad máxima se respetaba mediante el semáforo.

Pero en ocasiones, como lo indicó la parte de refinamiento del programa, habían ocasiones en que los usuarios pasaban un buen rato sin que el elevador llegara a su piso, por lo que (eventualmente, con muy mala suerte) podrían quedarse varados indefinidamente

Refinamiento: inanición

Si bien este potencial problema no detenía el programa (como en el caso de la primera versión) se podía abordar de otra forma:

Se añadieron dos banderas de turno (Event) que controlan qué grupo de pisos tiene prioridad en un determinado momento:

```
1 turno_bajo = threading.Event()
2 turno_alto = threading.Event()
3 turno_bajo.set() #Inicia por los pisos bajos
```

Los pisos se dividieron en dos grupos:

- Bajos: 0, 1, 2
- Altos: 3, 4

Así, el elevador alterna entre ambos grupos: cuando sube más allá del piso 2, activa el turno de pisos altos y cuando baja más allá del piso 3, regresa el turno a los pisos bajos.

Los usuarios verifican su grupo y esperan a que su turno sea activado:

```
1 grupo = 'bajo' if origen <= 2 else 'alto'
2 (turno_bajo if grupo == 'bajo' else turno_alto).wait()
```

Versión final

```

46         turno_alto.set()
47     elif piso_actual == 3 and direccion == -1:
48         turno_alto.clear()
49         turno_bajo.set() #Inicia por los pisos bajos
50
51 #Usuarios
52 def usuario(id, origen, destino):
53     global piso_actual
54     grupo = grupo_de_piso(origen)
55     turno = turno_bajo if grupo == 'bajo' else turno_alto
56     turno.wait() #El usuario espera turno
57     cond = cond_pisos[origen] #Esperar al elevador en el piso
    origen
58     with cond:
59         print(f"          Usuario {id} espera en piso {origen}
    para ir a {destino}")
60         cond.wait_for(lambda: piso_actual == origen)
61
62     capacidad.acquire() #El usuario intenta subirse
63     print(f"          Usuario {id} abord en piso {origen}")
64     with mutex_estado:
65         usuarios_en_elevador.append((id, destino))
66
67     while True:
68         time.sleep(0.5)
69         with mutex_estado:
70             if piso_actual == destino:
71                 print(f"          Usuario {id} baj en piso {
    destino}")
72                 usuarios_en_elevador.remove((id, destino))
73                 capacidad.release()
74                 break
75
76 #Se crean los hilos
77 threading.Thread(target=elevador, daemon=True).start()
78
79 for i in range(num_usuarios):
80     origen = random.randint(0, num_pisos-1)
81     destino = random.randint(0, num_pisos-1)
82     while destino == origen:
83         destino = random.randint(0, num_pisos-1)
84     threading.Thread(target=usuario, args=(i, origen, destino),
    daemon=True).start()
85     time.sleep(0.3)
86
87 #Para que no se detenga
88 while True:
89     time.sleep(2)

```



```
...▶ python ejerciciosSincElevador.py
👤 Usuario 0 espera en piso 3 para ir a 4
👤 Usuario 1 espera en piso 2 para ir a 4
👤 Usuario 2 espera en piso 1 para ir a 3
👤 Usuario 3 espera en piso 3 para ir a 1
👤 Usuario 4 espera en piso 4 para ir a 2

🟡 Elevador llegó al piso 1 (dir: ↑)
👤 Usuario 2 abordó en piso 1
👤 Usuario 5 espera en piso 2 para ir a 4
👤 Usuario 6 espera en piso 1 para ir a 3
👤 Usuario 6 abordó en piso 1
👤 Usuario 7 espera en piso 1 para ir a 3
👤 Usuario 7 abordó en piso 1
👤 Usuario 8 espera en piso 0 para ir a 3
👤 Usuario 9 espera en piso 2 para ir a 1

🟡 Elevador llegó al piso 2 (dir: ↑)
👤 Usuario 5 abordó en piso 2
👤 Usuario 9 abordó en piso 2
👤 Usuario 10 espera en piso 1 para ir a 3
👤 Usuario 11 espera en piso 0 para ir a 2

🟡 Elevador llegó al piso 3 (dir: ↑)
👤 Usuario 2 bajó en piso 3
👤 Usuario 1 abordó en piso 2
👤 Usuario 7 bajó en piso 3
👤 Usuario 0 abordó en piso 3
👤 Usuario 6 bajó en piso 3
👤 Usuario 3 abordó en piso 3

🟡 Elevador llegó al piso 4 (dir: ↓)
👤 Usuario 5 bajó en piso 4
👤 Usuario 4 abordó en piso 4
👤 Usuario 1 bajó en piso 4
👤 Usuario 0 bajó en piso 4

🟡 Elevador llegó al piso 3 (dir: ↓)

🟡 Elevador llegó al piso 2 (dir: ↓)
👤 Usuario 4 bajó en piso 2

🟡 Elevador llegó al piso 1 (dir: ↓)
👤 Usuario 10 abordó en piso 1
👤 Usuario 9 bajó en piso 1
👤 Usuario 3 bajó en piso 1

🟡 Elevador llegó al piso 0 (dir: ↑)
👤 Usuario 8 abordó en piso 0
```

Figure 2: Ejecución de la última versión con refinamiento.

Posibles mejoras

Si bien el planteamiento o reglas no lo especifica, se podrían hacer que los mismos usuarios repitan el uso del elevador o que se creen dinámicamente (que hagan spawn cada cuantos segundos o en intervalos random). Si bien se pueden crear tantos usuarios se quieran ajustando la variable global num usuarios, al final el programa se queda en un estado en el que el elevador se mueve de piso a piso, recorriendo los 5, pero ya no hay usuarios que se suban, por lo que creo que esta adición valdría la pena.