Proyecto Final - Mini Sistema Operativo Académico

Documentación del Proyecto

Integrantes:

Camacho Castelán José Manuel

López Castillo Haziel

Martha Denisse Lara Xocuis

Pichal Pío Jose Armando

Xalanda Coyolt Karina Arlet

H. Veracruz Ver. a 19 de junio de 2025

Índice

1 GestorPID.py

1.1 Descripción

Clase utilizada para gestionar de manera incremental los identificadores únicos de procesos (PID). Es estática y permite asegurar unicidad.

1.2 Código

```
1 class GestorPID:
2     siguiente_pid = 1
3 
4     @classmethod
5     def obtener_pid(cls):
6         pid = cls.siguiente_pid
7         cls.siguiente_pid += 1
8         return pid
```

Listing 1: GestorPID.py

1.3 Uso

Llamar a `GestorPID.obtener_pid()` devuelve el siguiente PID disponible.

2 Proceso.py

2.1 Descripción

Representa un proceso del sistema operativo simulado. Hereda de `Thread` y ejecuta una función en paralelo.

2.2 Código

```python
from threading import Thread
import time

class Proceso(Thread):
    def __init__(self, inicio, tamanio, func, pid):
        super().__init__()
        self.inicio = inicio
        self.tamanio = tamanio
        self.pid = pid
        self.func = func

    def run(self):
        if self.func:
            self.func(self.inicio, self.tamanio)
        else:
            print(f"[Proceso {self.pid}] No se asign  ninguna funci n.")

def tarea_ejemplo(inicio, tamanio):
    for i in range(5):
        print(f"[Proceso simulado] PID ocupando {tamanio} bloques desde {inicio}... ({i + 1}/5)")
        time.sleep(1)

def holamundo(inicio=None, tamanio=None):
    print("holamundo")
    time.sleep(1)
```

Listing 2: Proceso.py

2.3 Uso

El constructor recibe dirección inicial de memoria, tamaño, función a ejecutar y PID. Se ejecuta con `start()`.

3 Memoria.py

3.1 Descripción

Simula un espacio de memoria RAM donde se asignan bloques a procesos. Implementa el patrón Singleton.

3.2 Código

```
from kernel.Proceso import Proceso

class Memoria:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self, tamanio=100):
        if hasattr(self, '_initialized') and self._initialized:
            return
        self.tamanio = tamanio
        self.memoria = [0] * self.tamanio
        self.procesos = []
        self._initialized = True

    def asignar_memoria(self, pid, tamanio, func):
        libres = 0
        inicio = -1
        for i in range(len(self.memoria)):
            if self.memoria[i] == 0:
                if libres == 0:
                    inicio = i
                libres += 1
                if libres == tamanio:
                    for j in range(inicio, inicio + tamanio):
                        self.memoria[j] = pid
```

```
                    proceso = Proceso(inicio, tamanio, func, pid)
                    self.procesos.append(proceso)
                    print(f"Memoria asignada al proceso {pid} desde {
                        inicio} hasta {inicio + tamanio - 1}")
                    return proceso
            else:
                libres = 0
        print("No hay suficiente memoria contigua disponible.")
        return None

    def liberar_memoria(self, pid):
        liberados = 0
        for i in range(len(self.memoria)):
            if self.memoria[i] == pid:
                self.memoria[i] = 0
                liberados += 1
        self.procesos = [p for p in self.procesos if p.pid != pid]
        print(f"Se liberaron {liberados} bloques del proceso {pid}.")

    def __str__(self):
        s = "Estado de la memoria:\n"
        for i in range(0, self.tamanio, 20):
            fila = self.memoria[i:i + 20]
            s += " ".join([str(b) if b != 0 else "-" for b in fila]) +
                "\n"
        s += "Procesos:\n"
        for p in self.procesos:
            s += f"Pid: {p.pid}, Inicio: {p.inicio}\n"
        return s
```

Listing 3: Memoria.py

3.3 Uso

- `asignar_memoria(pid, tamanio, func)`: asigna bloques contiguos y crea el proceso.
- `liberar_memoria(pid)`: libera todos los bloques ocupados por el proceso con ese PID.
- `str(mem)`: devuelve una representación de texto de la memoria.