```
import os
import platform
import psutil
from typing import Union, Callable, List, Any
from multiprocessing import Process, Array
from ctypes import c_double
from loguru import logger



def set_cpu_affinity(cpu_id):
    try:
        if platform.system() == "Windows":
            import win32process


            win32process.SetProcessAffinityMask(
                os.getpid(), 1 << cpu_id
            )
        elif platform.system() == "Linux":
            os.sched_setaffinity(0, [cpu_id])
        else:
            logger.warning(
                f"CPU affinity not supported on {platform.system()}"
            )
    except Exception as e:
        logger.error(f"Failed to set CPU affinity: {e}")
```

```
def memory_intensive_task(size: int):
    large_list = [i for i in range(size)]
    return float(sum(large_list))



def task1():
    return memory_intensive_task(500000)



def task2():
    return memory_intensive_task(700000)



def task3():
    return memory_intensive_task(900000)



class FractionalizedCPU:
    def __init__(self):
        self.result_array = None

    def worker(
        self, func, index, cpu_id, memory_limit, *args, **kwargs
    ):
        try:
```

```
        # Set CPU affinity
        set_cpu_affinity(cpu_id)

        # Set memory limit
        try:
            import resource

            resource.setrlimit(
                resource.RLIMIT_AS, (memory_limit, memory_limit)
            )
        except ImportError:
            logger.warning(
                "resource module not available, memory limit not set"
            )

        result = func(*args, **kwargs)
        self.result_array[index] = result
    except Exception as e:
        logger.error(f"Error in worker process: {e}")
        self.result_array[index] = float("nan")

def execute_on_fractionalized_cpu(
    self,
    cpu_id: int,
    memory_fraction: float,
    func: Union[Callable, List[Callable]],
```

```
    *args: Any,
    **kwargs: Any,
) -> Any:
    """
    Executes a callable or list of callables on a fractionalized CPU core with limited memory.

    Args:
        cpu_id (int): The CPU core to run the function(s) on.
        memory_fraction (float): The fraction of the CPU's memory to allocate (0.0 to 1.0).
        func (Union[Callable, List[Callable]]): The function(s) to be executed.
        *args (Any): Arguments for the callable(s).
        **kwargs (Any): Keyword arguments for the callable(s).

    Returns:
        Any: The result(s) of the function execution(s).

    Raises:
        ValueError: If the CPU core specified is invalid or if the memory fraction is out of range.
        RuntimeError: If there is an error executing the function(s) on the CPU.
    """
    try:
        available_cpus = psutil.cpu_count(logical=False)
        if cpu_id < 0 or cpu_id >= available_cpus:
            raise ValueError(
                f"Invalid CPU core: {cpu_id}. Available CPUs are 0 to {available_cpus - 1}."
            )
```

```
if memory_fraction <= 0.0 or memory_fraction > 1.0:
    raise ValueError(
        "Memory fraction must be between 0.0 and 1.0."
    )

total_memory = psutil.virtual_memory().total
memory_limit = int(
    total_memory * memory_fraction / available_cpus
)

if isinstance(func, list):
    self.result_array = Array(c_double, len(func))
    processes = []
    for i, f in enumerate(func):
        p = Process(
            target=self.worker,
            args=(f, i, cpu_id, memory_limit) + args,
            kwargs=kwargs,
        )
        processes.append(p)
        p.start()

    for p in processes:
        p.join()
```

```
            results = list(self.result_array)
            return results
        else:
            self.result_array = Array(c_double, 1)
            p = Process(
                target=self.worker,
                args=(func, 0, cpu_id, memory_limit) + args,
                kwargs=kwargs,
            )
            p.start()
            p.join()

            return self.result_array[0]

    except Exception as e:
        logger.error(
            f"Error executing on fractionalized CPU {cpu_id}: {e}"
        )
        raise


# if __name__ == "__main__":
#     fractionalized_cpu = FractionalizedCPU()

#     # Execute a single function
#         result = fractionalized_cpu.execute_on_fractionalized_cpu(0, 0.1, memory_intensive_task,
```

```
1000000)
#     print(f"Single task result: {result}")


#     # Execute multiple functions
#     tasks = [task1, task2, task3]
#     results = fractionalized_cpu.execute_on_fractionalized_cpu(0, 0.3, tasks)
#     print(f"Multiple tasks results: {results}")
```