UNIVERSIDAD PERUANA
CAYETANO HEREDIA

Examen Parcial C8286

Reglas de la evaluación

- Queda terminantemente prohibido el uso de herramientas como ChatGPT, WhatsApp, o cualquier herramienta similar durante la realización de esta prueba. El uso de estas herramientas, por cualquier motivo, resultará en la anulación inmediata de la evaluación.
- Las respuestas deben presentarse con una explicación detallada, utilizando términos técnicos apropiados. La mera descripción sin el uso de terminología técnica, especialmente términos discutidos en clase, se considerará insuficiente y podrá resultar en que la respuesta sea marcada como incorrecta.
- Utiliza imágenes para explicar o complementar las respuestas, siempre y cuando estas sean de creación propia y relevantes para la respuesta, como es el caso de la verificación de los pasos de pruebas.
- Cada estudiante debe presentar su propio trabajo. Los códigos iguales o muy parecidos entre sí serán considerados como una violación a la integridad académica, implicando una copia, y serán sancionados de acuerdo con las políticas de la universidad.
- La claridad, orden, y presentación general de las evaluaciones serán tomadas en cuenta en la calificación final.

Instrucciones de entrega para la prueba calificada

Para asegurar una entrega adecuada y organizada de su prueba calificada, sigan cuidadosamente las siguientes instrucciones. El incumplimiento de estas pautas resultará en que su prueba no sea revisada, sin importar el escenario.

- Cada estudiante debe tener un repositorio personal en la plataforma designada para el curso (por ejemplo, GitHub, GitLab, etc.). El nombre del repositorio debe incluir su nombre completo o una identificación clara que permita reconocer al autor del trabajo fácilmente.
- Dentro de su repositorio personal, deben crear una carpeta titulada **ExamenParcial-C8286.** Esta carpeta albergará todas las soluciones de la prueba.
- Dentro de la carpeta **ExamenParcial-C8286,** deben crear dos subcarpetas adicionales, nombradas Pregunta1 **y** Pregunta2, respectivamente y dentro de cada parte Ejercicio1 y Ejercicio deben estar Ejercicio1, Ejercicio2, ..., Ejercicio7. Cada una de estas subcarpetas contendrá las soluciones y archivos correspondientes a cada ejercicio de la prueba.

  Ejemplo de estructura:
  /ExamenParcial-C8286
  /Pregunta1
  - Ejercicio1

- Ejercicio2
- Ejercico3

/Pregunta2

- Ejercicio1
- Ejercicio2
- Ejercicio3
- ...
- Ejercicio7

- Todas las respuestas y soluciones deben ser documentadas, completas y presentadas en archivos Markdown (.md) dentro de las subcarpetas correspondientes a cada ejercicio. Los archivos Markdown permiten una presentación clara y estructurada del texto y el código.
- No se aceptarán entregas en formatos como documentos de Word (.docx) o archivos PDF. La entrega debe cumplir estrictamente con el formato Markdown (.md) como se especifica.
- Es crucial seguir todas las instrucciones proporcionadas para la entrega de su prueba calificada. Cualquier desviación de estas pautas resultará en que su prueba no sea considerada para revisión.
- Antes de la entrega final, verifique la estructura de su repositorio, el nombramiento de las carpetas y archivos, y asegúrese de que toda su documentación esté correctamente formateada y completa.

Pregunta 1: Planificación y ejecución de tareas en sistemas paralelos y distribuidos

Sea el código llamado multi.py :

Importaciones y configuración inicial

```
from __future__ import print_function
from collections import namedtuple
from optparse import OptionParser
import random
```

Importaciones:

- **print_function** para compatibilidad entre Python 2 y 3.
- **namedtuple** de **collections** para definir una estructura de datos simple.
- **OptionParser** para manejar argumentos de línea de comandos.
- **random** para la generación de números aleatorios.

2. Semilla Aleatoria

```
def random_seed(seed):
   try:
      random.seed(seed, version=1)
   except:
      random.seed(seed)
   return
```

- **random_seed:** Inicializa el generador de números aleatorios con una semilla específica para asegurar reproducibilidad en los experimentos.

3. Función de ayuda para impresión

```
def print_cpu(cpu, str):
   print((' ' * cpu * 35) + str)
   return
```

- **print_cpu:** Imprime información de manera tabulada basada en el número de la CPU.

4. Definición de la estructura de trabajo

```
Job = namedtuple('Job', ['name', 'run_time', 'working_set_size', 'affinity', 'time_left'])
```

- **Job:** Define un trabajo con nombre, tiempo de ejecución, tamaño del conjunto de trabajo, afinidad de CPU y tiempo restante.

5. Clase Cache

```
class Cache:
    def __init__(self, cpu_id, jobs, cache_size, cache_rate_cold, cache_rate_warm,
cache_warmup_time):
        ...
    def new_job(self, job_name):
        ...
    def total_working_set(self):
        ...
    def adjust_size(self):
        ...
    def get_cache_state(self, job_name):
        ...
    def get_rate(self, job_name):
        ...
    def update_warming(self, job_name):
        ...
```

Cache: Simula una caché de CPU.

- **cpu_id:** ID de la CPU asociada.
- **jobs:** Diccionario de trabajos.
- **cache_size:** Tamaño de la caché.
- **cache_rate_cold y cache_rate_warm:** Velocidades de ejecución en frío y caliente.
- **cache_warmup_time:** Tiempo necesario para calentar la caché.
- **Métodos:** Manejan la inserción de trabajos en la caché, cálculo del conjunto de trabajo total, ajuste del tamaño de la caché, obtención del estado y la velocidad de la caché, y actualización del calentamiento.

6. Clase Scheduler

```
class Scheduler:
  def __init__(self, job_list, per_cpu_queues, affinity, peek_interval,
               job_num, max_run, max_wset,
               num_cpus, time_slice, random_order,
               cache_size, cache_rate_cold, cache_rate_warm, cache_warmup_time,
               solve, trace, trace_time_left, trace_cache, trace_sched):
    ...
  def handle_one_interrupt(self, interrupt, cpu):
    ...
  def handle_interrupts(self):
    ...
  def get_job(self, cpu, sched_queue):
    ...
  def assign_jobs(self):
    ...
  def print_sched_queues(self):
    ...
  def steal_jobs(self):
    ...
  def run_one_tick(self, cpu):
    ...
  def run_jobs(self):
    ...
  def run(self):
    ...
```

Scheduler: Simula un planificador de CPU múltiple.

- **job_list:** Lista de trabajos.
- **per_cpu_queues:** Indicador de si se utilizan colas por CPU o una cola central.
- **affinity:** Afinidad de trabajos a CPUs.
- **peek_interval:** Intervalo de tiempo para robar trabajos de otras colas.
- **job_num, max_run, max_wset:** Parámetros de trabajos.
- **num_cpus, time_slice, random_order:** Parámetros de CPU.
- **cache_size, cache_rate_cold, cache_rate_warm, cache_warmup_time:** Parámetros de caché.
- **solve, trace, trace_time_left, trace_cache, trace_sched:** Parámetros de trazado y solución.

7. Programa principal

```
parser = OptionParser()
parser.add_option('-s', '--seed',       default=0,     help='la semilla aleatoria', action='store', type='int', dest='seed')
parser.add_option('-j', '--job_num',    default=3,     help='número de trabajos en el sistema', action='store', type='int', dest='job_num')
parser.add_option('-R', '--max_run',    default=100,   help='tiempo máximo de ejecución de trabajos generados aleatoriamente', action='store', type='int', dest='max_run')
parser.add_option('-W', '--max_wset',   default=200,   help='máximo conjunto de trabajo de trabajos generados aleatoriamente', action='store', type='int', dest='max_wset')
parser.add_option('-L', '--job_list',   default='',    help='proporcionar una lista separada por comas de job_name:run_time:working_set_size (por ejemplo, a:10:100,b:10:50 significa 2 trabajos con tiempos de ejecución de 10, el primero (a) con tamaño de conjunto de trabajo=100, segundo (b) con tamaño de conjunto de trabajo=50)', action='store', type='string', dest='job_list')
parser.add_option('-p', '--per_cpu_queues', default=False, help='colas de planificación por CPU (no una)', action='store_true', dest='per_cpu_queues')
parser.add_option('-A', '--affinity',   default='',    help='una lista de trabajos y en qué CPUs pueden ejecutarse (por ejemplo, a:0.1.2,b:0.1 permite que el trabajo a se ejecute en las CPUs 0,1,2 pero b solo en las CPUs 0 y 1', action='store', type='string', dest='affinity')
parser.add_option('-n', '--num_cpus',   default=2,     help='número de CPUs', action='store', type='int', dest='num_cpus')
parser.add_option('-q', '--quantum',    default=10,    help='duración del intervalo de tiempo', action='store', type='int', dest='time_slice')
parser.add_option('-P', '--peek_interval', default=30,  help='para planificación por CPU, con qué frecuencia mirar en la cola de otro planificador; 0 desactiva esta función', action='store', type='int', dest='peek_interval')
parser.add_option('-w', '--warmup_time', default=10,    help='tiempo que lleva calentar la caché', action='store', type='int', dest='warmup_time')
parser.add_option('-r', '--warm_rate', default=2,     help='cuánto más rápido ejecutar con caché caliente', action='store', type='int', dest='warm_rate')
parser.add_option('-M', '--cache_size',  default=100,   help='tamaño de la caché', action='store', type='int', dest='cache_size')
parser.add_option('-o', '--rand_order',  default=False, help='hacer que las CPUs obtengan trabajos en orden aleatorio', action='store_true', dest='random_order')
parser.add_option('-t', '--trace',      default=False, help='habilitar trazado básico (mostrar qué trabajos fueron programados)', action='store_true', dest='trace')
parser.add_option('-T', '--trace_time_left', default=False, help='trazar el tiempo restante para cada trabajo', action='store_true', dest='trace_time_left')
parser.add_option('-C', '--trace_cache', default=False, help='trazar el estado de la caché (caliente/fría) también', action='store_true', dest='trace_cache')
parser.add_option('-S', '--trace_sched', default=False, help='trazar el estado del planificador', action='store_true', dest='trace_sched')
```

```
parser.add_option('-c', '--compute',    default=False, help='calcular respuestas para mí',
action='store_true', dest='solve')

(options, args) = parser.parse_args()

random_seed(options.seed)

print('ARG seed %s' % options.seed)
print('ARG job_num %s' % options.job_num)
print('ARG max_run %s' % options.max_run)
print('ARG max_wset %s' % options.max_wset)
print('ARG job_list %s' % options.job_list)
print('ARG affinity %s' % options.affinity)
print('ARG per_cpu_queues %s' % options.per_cpu_queues)
print('ARG num_cpus %s' % options.num_cpus)
print('ARG quantum %s' % options.time_slice)
print('ARG peek_interval %s' % options.peek_interval)
print('ARG warmup_time %s' % options.warmup_time)
print('ARG cache_size %s' % options.cache_size)
print('ARG random_order %s' % options.random_order)
print('ARG trace %s' % options.trace)
print('ARG trace_time %s' % options.trace_time_left)
print('ARG trace_cache %s' % options.trace_cache)
print('ARG trace_sched %s' % options.trace_sched)
print('ARG compute %s' % options.solve)
print('')

#
# TRABAJOS
#
job_list = options.job_list
job_num = int(options.job_num)
max_run = int(options.max_run)
max_wset = int(options.max_wset)

#
# MÁQUINA
#
num_cpus = int(options.num_cpus)
time_slice = int(options.time_slice)

#
# CACHÉS
```

```
#
cache_size = int(options.cache_size)
cache_rate_warm = int(options.warm_rate)
cache_warmup_time = int(options.warmup_time)

do_trace = options.trace
if options.trace_time_left or options.trace_cache or options.trace_sched:
   do_trace = True

#
# PLANIFICADOR (y simulador)
#
S = Scheduler(job_list=job_list, affinity=options.affinity, per_cpu_queues=options.per_cpu_queues,
peek_interval=options.peek_interval,
        job_num=job_num, max_run=max_run, max_wset=max_wset,
        num_cpus=num_cpus, time_slice=time_slice, random_order=options.random_order,
        cache_size=cache_size, cache_rate_cold=1, cache_rate_warm=cache_rate_warm,
        cache_warmup_time=cache_warmup_time, solve=options.solve,
        trace=do_trace, trace_time_left=options.trace_time_left, trace_cache=options.trace_cache,
        trace_sched=options.trace_sched)

# Finalmente, ...
S.run()
```

Parser de opciones: Define y parsea las opciones de línea de comandos para personalizar la simulación.

- **Inicialización de semilla:** Inicializa la semilla aleatoria.
- **Configuración de trabajos, máquina y cachés:** Configura los parámetros de la simulación.
- **Instancia del planificador:** Crea una instancia del planificador con los parámetros configurados.
- **Ejecución de la simulación:** Ejecuta la simulación llamando al método **run** del planificador.

Ejercicio 1: Puedes utilizar estos items para analizar el código entregado (2 puntos)

1. La primera simulación ejecutará solo un trabajo, que tiene un tiempo de ejecución de 30 y un tamaño de conjunto de trabajo de 200. Ejecuta este trabajo (llamado trabajo 'a' aquí) en una CPU simulada de la siguiente manera: **./multi.py -n 1 -L a:30:200**. ¿Cuánto tiempo tomará completarse? Activa la bandera **-c** para ver una respuesta final, y la bandera **-t** para ver un rastro del trabajo paso a paso y cómo se programa.
2. Ahora aumenta el tamaño de la caché para que el conjunto de trabajo del trabajo (tamaño=200) quepa en la caché (que, por defecto, tiene un tamaño de 100); por ejemplo, ejecuta **./multi.py -n 1 -L a:30:200 -M 300**. ¿Puedes predecir qué tan rápido se ejecutará el trabajo una vez que quepa en la caché? (pista: recuerda el parámetro clave de la tasa de

calentamiento, que se establece con la bandera **-r**). Verifica tu respuesta ejecutando con la bandera de solución (**-c**) activada.

3. Una cosa interesante sobre **multi.py** es que puedes ver más detalles sobre lo que está ocurriendo con diferentes banderas de rastreo. Ejecuta la misma simulación que antes, pero esta vez con el rastreo de tiempo restante habilitado (**-T**). Esta bandera muestra tanto el trabajo que se programó en una CPU en cada paso de tiempo, como el tiempo de ejecución que le queda a ese trabajo después de cada tick. ¿Qué notas sobre cómo disminuye esa segunda columna?
4. Ahora agrega un poco más de rastreo, para mostrar el estado de cada caché de CPU para cada trabajo, con la bandera **-C**. Para cada trabajo, cada caché mostrará un espacio en blanco (si la caché está fría para ese trabajo) o una 'w' (si la caché está caliente para ese trabajo). ¿En qué momento se calienta la caché para el trabajo 'a' en este ejemplo simple? ¿Qué sucede al cambiar el parámetro de tiempo de calentamiento (**-w**) a valores menores o mayores que el predeterminado?

Ejercicio2 Compara la planificación centralizada y distribuida en términos de rendimiento y utilización de la CPU. Modifica el código para permitir la simulación con una cola centralizada y colas distribuidas (4 puntos)

Implementaciones:

- Configura dos simulaciones: una con una cola centralizada y otra con colas distribuidas.
- Ejecuta ambas simulaciones con el mismo conjunto de trabajos y configura varias CPUs.
- Reporta el tiempo total de ejecución y la utilización de la CPU para cada enfoque.
- Analiza los resultados y discute las ventajas y desventajas de cada enfoque en diferentes escenarios.

Pregunta 2: El paradigma de Map-reduce

El código siguiente permite manejar trabajos de MapReduce de manera distribuida y asíncrona, utilizando colas y múltiples procesos para mejorar la eficiencia en el procesamiento de grandes volúmenes de datos.

```
import asyncio
import marshal
import multiprocessing as mp
import pickle
from queue import Empty, Queue  # PriorityQueue
import threading
import types

import chunk_mp_mapreduce as mr


# multiprocessing.Queue
```


```
work_queue = Queue()
results_queue = Queue()
results = {}

async def submit_job(job_id, reader, writer):
    writer.write(job_id.to_bytes(4, 'little'))
    writer.close()
    code_size = int.from_bytes(await reader.read(4), 'little')
    my_code = marshal.loads(await reader.read(code_size))
    data_size = int.from_bytes(await reader.read(4), 'little')
    data = pickle.loads(await reader.read(data_size))
    work_queue.put_nowait((job_id, my_code, data))


def get_results_queue():
    while results_queue.qsize() > 0:
        try:
            job_id, data = results_queue.get_nowait()
            results[job_id] = data
        except Empty:
            return


async def get_results(reader, writer):
    get_results_queue()
    job_id = int.from_bytes(await reader.read(4), 'little')
    data = pickle.dumps(None)
    if job_id in results:
        data = pickle.dumps(results[job_id])
        del results[job_id]
    writer.write(len(data).to_bytes(4, 'little'))
    writer.write(data)


async def accept_requests(reader, writer, job_id=[0]):
    op = await reader.read(1)
    if op[0] == 0:
        await submit_job(job_id[0], reader, writer)  #Errors in async
        job_id[0] += 1
    elif op[0] == 1:
        await get_results(reader, writer)
```


```
def worker():  # daemon
    pool = mp.Pool()
    while True:
        job_id, code, data = work_queue.get()  # blocking
        func = types.FunctionType(code, globals(), 'mapper_and_reducer')
        mapper, reducer = func()
        counts = mr.map_reduce(pool, data, mapper, reducer, 100, mr.reporter)
        results_queue.put((job_id, counts))
    pool.close()
    pool.join()


async def main():
    server = await asyncio.start_server(accept_requests, '127.0.0.1', 1936)
    worker_thread = threading.Thread(target=worker)   # Daemon
    worker_thread.start()
    async with server:
        await server.serve_forever()


asyncio.run(main())
```

Los problemas solo implican mejorar el código base entregado.

Ejercicio 1: Mejora del Manejo de errores y loggin (1 punto)

Mejora el código existente añadiendo manejo de errores robusto y logging para depuración y monitoreo.

Implementaciones:

1. Añade manejo de errores en todas las funciones asíncronas (**submit_job**, **get_results**, **accept_requests**).
2. Integrar el módulo **logging** para registrar información relevante, advertencias y errores.
3. Asegura de que todas las excepciones sean manejadas adecuadamente y no causen la terminación inesperada del programa.

Ejercicio 2: Refactorización del trabajador (1 punto)

Refactorizar la función **worker** para utilizar **concurrent.futures** en lugar de **multiprocessing.Pool**.

Implementaciones:

Reemplazar el uso de **multiprocessing.Pool** con **concurrent.futures.ProcessPoolExecutor**.

1. Asegurarse de que el nuevo enfoque mantenga la misma funcionalidad y eficiencia.

2. Añade pruebas para verificar que los trabajos de MapReduce se ejecutan correctamente con el nuevo enfoque.

Ejercicio 3: Optimización de la comunicación asíncrona (1 puntos)

Optimiza la comunicación asíncrona utilizando mejores prácticas de **asyncio**.

Implementaciones

- Optimiza el uso de **asyncio.Queue** en lugar de **queue.Queue** para **work_queue** y **results_queue**.
- Asegurate de que todas las operaciones de cola sean asíncronas.
- Revisar y mejorar el rendimiento del manejo de conexiones y transferencia de datos.

Ejercicio 4: Contenerización con Docker (2 puntos)

Crea un Dockerfile para contenerizar la aplicación.

Implementaciones:

- Escribe un **Dockerfile** para construir una imagen de Docker que ejecute la aplicación.
- Asegurate de que la imagen sea ligera y eficiente.
- Prueba la imagen de Docker para verificar que la aplicación funciona correctamente dentro del contenedor.

Ejercicio 5: Despliegue en Kubernetes (2 puntos)

Despliega la aplicación en un clúster de Kubernetes.

Implementaciones:

- Escribie archivos de configuración de Kubernetes (**Deployment**, **Service**) para desplegar la aplicación.
- Asegurate de que la aplicación esté correctamente escalada y manejada por Kubernetes.
- Implementa monitoreo y logging para el despliegue de Kubernetes.

Ejercicio 6: Implementación de funcionalidades avanzadas de mapReduce (2 puntos)

Añadir funcionalidades avanzadas al paradigma MapReduce implementado.

Tareas:

- Añae soporte para combinadores (combiner) que permitan reducir el volumen de datos en la fase de mapeo.
- Implementa particionadores personalizados para controlar cómo se distribuyen las tareas de reducción.
- Añade soporte para operaciones de reducción en varios pasos (multi-step reduce).

Ejercicio 7: Mejores al código para serlo más robusto (5 puntos)

- ¿Cómo podrías asegurarte de que el servidor se cierra correctamente al recibir una señal de interrupción (por ejemplo, Ctrl+C)? Implementa un manejador de señales para limpiar recursos y cerrar el servidor de manera ordenada.
- ¿Cómo podrías utilizar funciones parciales para mejorar la legibilidad y la gestión de parámetros en el código? Utiliza **functools.partial** para simplificar la creación de funciones con múltiples parámetros predefinidos.
- **:**¿Cómo puedes inicializar adecuadamente los trabajadores y gestionar un pool de procesos, asegurando que se ignoren ciertas señales en los trabajadores? Utiliza **mp.Pool** con un inicializador que configure la gestión de señales apropiadamente en los procesos hijos.
- ¿Cómo aseguras que los trabajadores terminan correctamente cuando se cierra el servidor? Envia un trabajo especial (sentinela) a los trabajadores para indicarles que deben finalizar. No te olvides de que los hilos trabajadores se unen adecuadamente antes de que el programa principal termine.
- ¿Cómo manejas adecuadamente las excepciones y errores para asegurarte de que el servidor no falle de manera inesperada? Captura y maneja las excepciones en las funciones asincrónicas y en los hilos trabajadores. Utiliza bloques **try-except** para manejar errores en la comunicación entre el cliente y el servidor.
- ¿Cómo puedes mejorar la estructura general y la legibilidad del código para hacerlo más mantenible y robusto?. Refactoriza el código para separar claramente las responsabilidades (por ejemplo, manejar trabajos, manejar resultados, aceptar solicitudes). Utiliza nombres de variables y funciones descriptivos para mejorar la claridad del código.