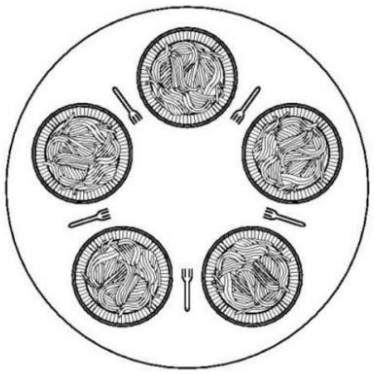

CENA DE LOS FILÓSOFOS

INFORME ACTIVIDAD GRUPAL 1

Programación concurrente y distribuida

Eduardo CASTELLÓN RAMÍREZ
Camilo BETANCUR
Ángel ESCOBAR ANCHUELO
Marcos Eladio SOMOZA CORRAL
Eliseo NGUEMA CHEMA

Índice

Resumen del problema planteado

Cinco filósofos se sientan alrededor de una mesa para pensar y comer. Cada filósofo tiene un plato delante suyo, y hay un tenedor entre plato y plato. Cuando un filósofo para de pensar y quiere comer, necesita coger los dos tenedores que están al lado de su plato.

Se asume que cada filósofo tiene comida infinita. Cada tenedor se coge y se suelta por separado.

Problemas

- Si dos filósofos que están uno al lado del otro, quieren coger el mismo tenedor a la vez, se producirá una condición de carrera: ambos competirán por coger el mismo tenedor y uno de ellos se quedará sin comer.
- Si cada filósofo coge el tenedor de su izquierda y espera a que el de su derecha esté libre, caemos en un problema de interbloqueo porque todos se quedarán esperando eternamente y se morirán de hambre.

Posibles soluciones

- Se podría hacer que los filósofos esperaran un tiempo arbitrario en vez del mismo tiempo, de esta forma sería poco probable que todos quedaran bloqueados. Esta solución no es recomendada.
- Presencia de un vector estado para cada filósofo. De esta forma, se podría llevar un registro (comiendo, pensando o hambriento) indicando si quiere o no coger los tenedores. Un filósofo sólo puede comer si sus compañeros adyacentes no están comiendo.
- Utilizar un semáforo por filósofo para que los que están hambrientos puedan bloquearse si los tenedores necesarios están ocupados.

Soluciones que evitan el interbloqueo mediante semáforos

- Introducción de un concepto camarero que se implementa como semáforo
- Que hubiera un orden entre los tenedores (sólo se pueden coger y liberar en orden).
- Solución de Chandy y Misra que requeriría comunicación entre procesos, pero esto no se contempla en el problema inicial.
- Solución con monitores pero puede provocar inanición.

Se ha implementado en pseudocódigo la solución al problema original, así como la solución con vectores y semáforos comentada anteriormente.

Pseudocódigo

```
/* SOLUCIÓN ORIGINAL */
#define N 5 /* número de filósofos */
void filosofo(int i) {
while(true) {
pensar(); /* está pensando */
coger_tenedor(i); /* coge el tenedor izquierdo */
coger_tenedor((i + 1) % N) /* coge el tenedor derecho */
comer(); /* come */
dejar_tenedor(i); /* deja el tenedro izquierdo */
dejar_tenedor((i + 1 ) % N) /* deja el tenedor derecho */
}
}

/* SOLUCIÓN CON VECTORES Y SEMÁFOROS. SECCIÓN CRÍTICA */
#define N 5 /* número de filósofos */
#define IZQ (i - 1) % N /* num del adyacente izq. de i */
#define DER (i + 1) % N /* num del adyacente der. de i */
#define PENSANDO 0 /* está pensando */
#define HAMBRIENTO 1 /* está hambriento */
#define COMIENDO 2 /* está comiendo */
int estado[N]; /* vector estado de filósofos */
semaforo mutex = 1; /* mutex para las secciones críticas */
semaforo s[N]; /* un semáforo por cada filósofo */
void filosofo(int i) { /* i es el num de filosofo */
while(true) {
pensar(); /* está pensando */
coger_tenedores(); /* obtiene 2 tenedores, se bloquea si
no puede */
comer(); /* está comiendo*/
dejar_tenedores(); /* deja los 2 tenedores en la mesa */
}
}
void coger_tenedores(int i) {
```

```
wait(mutex); /* entra en la seccion crítica */
estado[i] = HAMBRIENTO; /* el filósofo i tiene hambre */
prueba(i); /* intenta coger los dos tenedores */
signal(mutex); /* sale de la sección crítica */
wait(s[i]); /* se bloquea si no consiguió los
dos tenedores */

}
void dejar_tenedores(int i) {
wait(mutex); /* entra en la sección crítica */
estado[i] = PENSANDO; /* el filósofo i ha dejado de comer */
prueba(IZQ); /* comprueba si el adyacente por la izq
puede comer ahora */

prueba(DER); /* comprueba si el adyacente por la der
puede comer ahora */
signal(mutex); /* sale de la sección crítica */
}
void prueba(int i) {
if(estado[i] == HABRIENTO && estado[IZQ] != COMIENDO
&& estado[DER] != COMIENDO) {
estado[i] = COMIENDO; /* el filósofo i está comiendo */
signal(s[i]);
}
}
```

Código en Python

```
# Symmetric solution to the "dining philosophers"
# problem. Uses a semaphore as the "butler" to avoid
# deadlock.

import sys
import threading
import time

class Semaphore(object):

    def __init__(self, initial): # initial method
        self.lock = threading.Condition(threading.Lock()) # thread lock c
ondition
        self.value = initial

    def up(self):  # new philosopher sits
        with self.lock:
            self.value += 1
            self.lock.notify()

    def down(self):  # philosopher gets up
        with self.lock:
            while self.value == 0:
                self.lock.wait()
            self.value -= 1

class ChopStick(object):

    def __init__(self, number):
        self.number = number           # chop stick ID
        self.user = -
1                # keep track of philosopher using it
        self.lock = threading.Condition(threading.Lock()) # thread lock c
ondition
        self.taken = False # this fork is now available

    def take(self, user):         # used for synchronization, philosopher
 takes fork
        with self.lock:
            while self.taken == True: # while this fork is taken, wait
                self.lock.wait()
            self.user = user # the philosopher that has this fork
            self.taken = True # this fork is not available
            sys.stdout.write("p[%s] took c[%s]\n" % (user, self.number))
            self.lock.notifyAll() # notifies every thread
```

```
    def drop(self, user):          # used for synchronization, philosopher
 drops fork
        with self.lock:
            while self.taken == False: # while this fork is free, wait
                self.lock.wait()
            self.user = -1 # none philosopher that has this fork
            self.taken = False # this fork is now available
            sys.stdout.write("p[%s] dropped c[%s]\n" % (user, self.number
))
            self.lock.notifyAll() # notifies every thread


class Philosopher (threading.Thread):

    def __init__(self, number, left, right, butler): #init method
        threading.Thread.__init__(self) #init thead to current philosophe
r
        self.number = number            # philosopher number
        self.left = left                # assing left fork
        self.right = right              # assing right fork
        self.butler = butler            # assing butler

    def run(self):
        for i in range(20):
            self.butler.down()              # start service by butler
            time.sleep(0.1)                 # think
            self.left.take(self.number)     # pickup left chopstick
            time.sleep(0.1)                 # (yield makes deadlock more
likely)
            self.right.take(self.number)    # pickup right chopstick
            time.sleep(0.1)                 # eat
            self.right.drop(self.number)    # drop right chopstick
            self.left.drop(self.number)     # drop left chopstick
            self.butler.up()                # end service by butler
        sys.stdout.write("p[%s] finished thinking and eating\n" % self.nu
mber)


def main():
    # number of philosophers / chop sticks
    n = 5

    # butler for deadlock avoidance (n-1 available)
    butler = Semaphore(n-1)

    # list of chopsticks
    c = [ChopStick(i) for i in range(n)]
```

```
    # list of philsophers
    p = [Philosopher(i, c[i], c[(i+1)%n], butler) for i in range(n)]

    for i in range(n): # foreach philosopher in array -> start thread
        p[i].start()


if __name__ == "__main__":
    main()
```

Github´s

Eduardo CASTELLÓN RAMÍREZ - https://github.com/Edu-Castellon/Concurrentes

Camilo BETANCUR - https://github.com/Camiwi/Concurrente

Ángel ESCOBAR ANCHUELO - https://github.com/SecretoAngel/Concurrente

Marcos Eladio SOMOZA CORRAL- - https://github.com/somozadev/21711787AG1_CenaFilosofos

Eliseo NGUEMA CHEMA-