Problema de los Filósofos Comensales

Resumen

Este proyecto presenta una implementación completa del problema clásico de sincronización de los Filósofos Comensales en tres lenguajes de programación: C, Java y Python. Cada implementación incluye soluciones basadas en hilos (threads) y procesos independientes, demostrando diferentes mecanismos de sincronización y comunicación entre procesos (IPC).

Descripción del Problema

El problema de los Filósofos Comensales, propuesto por Edsger Dijkstra en 1965, es un problema clásico de sincronización en sistemas concurrentes. El escenario consiste en N filósofos sentados alrededor de una mesa circular, donde cada filósofo alterna entre dos estados: pensar y comer. Entre cada par de filósofos hay exactamente un tenedor, resultando en N tenedores en total. Para comer, un filósofo requiere adquirir ambos tenedores adyacentes (izquierdo y derecho).

Los principales desafíos de este problema son:

1. **Deadlock**: Evitar que todos los filósofos tomen simultáneamente su tenedor izquierdo, bloqueándose mutuamente al esperar el tenedor derecho.
2. **Starvation**: Garantizar que ningún filósofo permanezca indefinidamente sin poder comer.
3. **Concurrencia**: Maximizar el paralelismo permitiendo que múltiples filósofos coman simultáneamente cuando sea posible.

Estrategia de Prevención de Deadlock y Starvation

Algoritmo Implementado

Todas las implementaciones utilizan una estrategia centralizada de control basada en el algoritmo de "verificación de vecinos". La estrategia consiste en:

1. **Control Centralizado**: Una entidad central (mesa) coordina el acceso a los recursos compartidos (tenedores).
2. **Verificación de Vecinos**: Un filósofo solo puede tomar ambos tenedores simultáneamente si:
   - El filósofo está en estado HAMBRIENTO
   - El vecino izquierdo NO está COMIENDO
   - El vecino derecho NO está COMIENDO
3. **Notificación Proactiva**: Cuando un filósofo libera sus tenedores:
   - Cambia su estado a PENSANDO
   - La mesa verifica si los vecinos adyacentes pueden ahora comer
   - Se notifica explícitamente a los vecinos elegibles

4. **Atomicidad en Adquisición**: Los tenedores se adquieren como una operación atómica, evitando estados intermedios que puedan causar deadlock.

Esta estrategia previene el deadlock al evitar la adquisición secuencial de recursos y previene la starvation mediante notificaciones explícitas que garantizan que los filósofos bloqueados sean despertados cuando los recursos estén disponibles.

Implementación en C

Estructura del Proyecto

```
filosofos_c/
|-- Makefile
|-- bin/
|   |-- filosofos           # Ejecutable con hilos POSIX
|   +-- filosofos_procesos  # Ejecutable con procesos fork()
|-- obj/                    # Archivos objeto compilados
+-- src/
    |-- hilos/             # Implementacion con threads
    |   |-- main.c
    |   |-- filosofo.c/h
    |   |-- mesa.c/h
    |   +-- tenedor.c/h
    +-- procesos/          # Implementacion con fork()
        |-- main.c
        |-- proceso_filosofo.c/h
        +-- mesa_ipc.c/h
```

Mecanismos de Sincronización

Versión con Hilos: - **Mutex Global** (`pthread_mutex_t`): Protege el acceso a la tabla de estados compartida - **Variable de Condición** (`pthread_cond_t`): Implementa el mecanismo de espera y notificación - **Broadcast**: Utiliza `pthread_cond_broadcast()` para notificar a todos los filósofos esperando

Versión con Procesos: - **Memoria Compartida** (`mmap` con `MAP_SHARED`): Tabla de estados y contadores - **Semáforos POSIX** (`sem_t`): Array de semáforos para sincronización entre procesos - **Semáforo Global**: Protege las secciones críticas en operaciones de la mesa

Compilación y Ejecución

```
# Compilar
cd filosofos_c
make clean
make

# Ejecutar versión con hilos
```

```
./bin/filosofos [num_filosofos] [duracion_segundos]
./bin/filosofos 5 30

# Ejecutar versión con procesos
./bin/filosofos_procesos [num_filosofos] [duracion_segundos]
./bin/filosofos_procesos 5 30
```

Resultados Experimentales

Configuración de prueba: 5 filósofos, 30 segundos de ejecución

Versión con Hilos:

```
Filósofo 0 comió 11 veces
Filósofo 1 comió 11 veces
Filósofo 2 comió 9 veces
Filósofo 3 comió 12 veces
Filósofo 4 comió 13 veces

Total de veces que se comió: 56
Promedio por filósofo: 11.20
Desviación estándar: 1.47
```

Versión con Procesos:

```
Filósofo 0 comió 11 veces
Filósofo 1 comió 11 veces
Filósofo 2 comió 12 veces
Filósofo 3 comió 10 veces
Filósofo 4 comió 10 veces

Total de veces que se comió: 54
Promedio por filósofo: 10.80
Desviación estándar: 0.84
```

Análisis: Ambas versiones demuestran una distribución equitativa de acceso a recursos, con desviaciones estándar bajas que indican ausencia de starvation. La ligera variación en el total se debe a diferencias en el overhead de sincronización entre hilos y procesos.

Implementación en Java

Estructura del Proyecto

```
filosofos_java/
|-- bin/                    # Archivos .class compilados
+-- src/
    |-- EjecutarHilos.java
    |-- EjecutarProcesos.java
```

```
|-- hilos/            # Implementacion con Threads
|   |-- Filosofo.java
|   |-- Mesa.java
|   |-- Tenedor.java
|   +-- Estado.java
|-- procesos/         # Simulacion de procesos con Threads
|   |-- ProcesoFilosofo.java
|   |-- MesaIPC.java
|   +-- EstadoFilosofo.java
+-- procesosreales/   # Procesos JVM independientes
    |-- FilosofoClient.java
    |-- MesaServer.java
    |-- Launcher.java
    +-- Estado.java
```

Mecanismos de Sincronización

Versión con Hilos: - **ReentrantLock**: Lock explícito para control de acceso mutuo - **Condition**: Mecanismo de espera y señalización asociado al lock - **signalAll()**: Notificación broadcast a todos los hilos en espera

Versión con Procesos Simulados: - **Semaphore**: Implementación de semáforos de Java para control de acceso - **AtomicIntegerArray**: Estructura atómica para tabla de estados compartida - **Array de Semaphores**: Uno por filósofo para bloqueo individual

Versión con Procesos Reales: - **Sockets TCP**: Comunicación entre JVMs independientes - **Arquitectura Cliente-Servidor**: MesaServer coordina múltiples FilosofoClient - **ProcessBuilder**: Creación de procesos Java separados

Compilación y Ejecución

```
# Compilar
cd filosofos_java
javac -d bin src/*.java src/hilos/*.java src/procesos/*.java src/procesosreales/*.java

# Ejecutar versión con hilos
java -cp bin EjecutarHilos [num_filosofos] [duracion_segundos]
java -cp bin EjecutarHilos 5 30

# Ejecutar versión con procesos simulados
java -cp bin EjecutarProcesos [num_filosofos] [duracion_segundos]
java -cp bin EjecutarProcesos 5 30

# Ejecutar versión con procesos reales (múltiples JVMs)
java -cp bin procesosreales.Launcher [num_filosofos] [duracion_segundos]
java -cp bin procesosreales.Launcher 5 30
```

Resultados Experimentales

Configuración de prueba: 5 filósofos, 30 segundos de ejecución

Versión con Hilos:

```
Filósofo 0 comió 10 veces
Filósofo 1 comió 11 veces
Filósofo 2 comió 10 veces
Filósofo 3 comió 9 veces
Filósofo 4 comió 11 veces

Total de veces que se comió: 51
Promedio por filósofo: 10.20
Desviación estándar: 0.84
```

Versión con Procesos Simulados:

```
Filósofo 0 comió 9 veces
Filósofo 1 comió 10 veces
Filósofo 2 comió 11 veces
Filósofo 3 comió 10 veces
Filósofo 4 comió 9 veces

Total de veces que se comió: 49
Promedio por filósofo: 9.80
Desviación estándar: 0.84
```

Versión con Procesos Reales:

```
Filósofo 0 comió 8 veces
Filósofo 1 comió 9 veces
Filósofo 2 comió 9 veces
Filósofo 3 comió 8 veces
Filósofo 4 comió 8 veces

Total de veces que se comió: 42
Promedio por filósofo: 8.40
Desviación estándar: 0.55
```

Análisis: La versión con procesos reales muestra un rendimiento ligeramente inferior debido al overhead de comunicación por sockets TCP. Sin embargo, la distribución sigue siendo equitativa, demostrando la efectividad del algoritmo independientemente del mecanismo de IPC utilizado.

Implementación en Python

Estructura del Proyecto

```
filosofos_python/
```

```
|-- ejecutar_hilos.py
|-- ejecutar_procesos.py
|-- solucion_hilos/       # Implementacion con threading
|   |-- filosofo.py
|   |-- mesa.py
|   +-- tenedor.py
+-- solucion_procesos/    # Implementacion con multiprocessing
    |-- ProcesoFilosofo.py
    +-- MesaIPC.py
```

Mecanismos de Sincronización

Versión con Hilos: - **threading.Lock**: Mutex para protección de secciones críticas - **threading.Condition**: Variable de condición para espera y notificación - **notify_all()**: Notificación broadcast a todos los hilos bloqueados

Versión con Procesos: - **multiprocessing.Semaphore**: Semáforos para sincronización entre procesos - **multiprocessing.Array**: Array compartido para tabla de estados - **multiprocessing.Lock**: Lock para proteger operaciones críticas

Compilación y Ejecución

Python no requiere compilación. Las implementaciones se ejecutan directamente:

```
# Ejecutar versión con hilos
cd filosofos_python
python ejecutar_hilos.py [num_filosofos] [duracion_segundos]
python ejecutar_hilos.py 5 30

# Ejecutar versión con procesos
python ejecutar_procesos.py [num_filosofos] [duracion_segundos]
python ejecutar_procesos.py 5 30
```

Resultados Experimentales

Configuración de prueba: 5 filósofos, 30 segundos de ejecución

Versión con Hilos:

```
Filósofo 0 comió 12 veces
Filósofo 1 comió 11 veces
Filósofo 2 comió 10 veces
Filósofo 3 comió 11 veces
Filósofo 4 comió 12 veces

Total de veces que se comió: 56
Promedio por filósofo: 11.20
Desviación estándar: 0.84
```

Versión con Procesos:

```
Filósofo 0 comió 10 veces
Filósofo 1 comió 11 veces
Filósofo 2 comió 10 veces
Filósofo 3 comió 10 veces
Filósofo 4 comió 11 veces

Total de veces que se comió: 52
Promedio por filósofo: 10.40
Desviación estándar: 0.55
```

Análisis: Python muestra resultados consistentes con las otras implementaciones. La versión con hilos presenta un rendimiento ligeramente superior debido al Global Interpreter Lock (GIL) que reduce el overhead en cambios de contexto, aunque limita el paralelismo real en sistemas multi-core.

Comparación de Rendimiento

Tabla Comparativa

Implementación	Total Comidas	Promedio	Desv. Estándar	Overhead
C - Hilos	56	11.20	1.47	Bajo
C - Procesos	54	10.80	0.84	Medio
Java - Hilos	51	10.20	0.84	Medio
Java - Procesos	49	9.80	0.84	Medio
Java - Reales	42	8.40	0.55	Alto
Python - Hilos	56	11.20	0.84	Bajo
Python - Procesos	52	10.40	0.55	Medio

Observaciones

1. **Rendimiento**: Las implementaciones en C muestran el mayor número de operaciones completadas, seguidas por Python y Java.
2. **Equidad**: Todas las implementaciones demuestran distribuciones equitativas (desviaciones estándar < 1.5), validando la ausencia de starvation.
3. **Overhead**: Los mecanismos basados en procesos presentan mayor overhead que los basados en hilos, especialmente en la versión con procesos reales de Java que requiere comunicación por red.
4. **Consistencia**: La estrategia de verificación de vecinos produce resultados consistentes independientemente del lenguaje o mecanismo de sincronización utilizado.

Conclusiones

Este proyecto demuestra exitosamente la implementación del algoritmo de verificación de vecinos para resolver el problema de los Filósofos Comensales. Las principales conclusiones son:

1. **Corrección**: Todas las implementaciones evitan tanto deadlock como starvation mediante control centralizado y verificación atómica de condiciones.
2. **Portabilidad**: El algoritmo es efectivo independientemente del lenguaje de programación y mecanismo de sincronización utilizado.
3. **Escalabilidad**: Las soluciones mantienen su corrección y equidad con diferentes números de filósofos (probado con N = 3, 5, 7).
4. **Trade-offs**: Existe un compromiso observable entre el overhead de comunicación y el aislamiento de recursos, siendo los hilos más eficientes pero los procesos más robustos ante fallos.

Referencias

1. Dijkstra, E. W. (1971). "Hierarchical ordering of sequential processes". Acta Informatica, 1(2), 115-138.
2. Tanenbaum, A. S., & Bos, H. (2014). Modern Operating Systems (4th ed.). Pearson.
3. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.