Pamela Pupiales 213871, Antonio Altamirano 210849, Alejandra Ospina 212243 y Camila Zambrano 209260
Sistemas Operativos
Daniel Riofrío

Tarea 2

Ejercicio 1: En el trabajo que han realizado referente a la simulación de algoritmos de calendarización, en este ejercicio, ustedes van a completar su simulador considerando.

1. **Una revisión a la lista de requerimientos que construyeron en sus equipos de trabajo.**

- **Configuración de algoritmos:** El simulador debe permitir la configuración de diferentes algoritmos de calendarización. En nuestro caso, Round Robin, FCFS y SJF, los usuarios pueden seleccionar el algoritmo que deseen probar.
- **Modificación de los parámetros del proceso:** Esto permitirá al usuario experimentar con diferentes configuraciones para determinar la configuración óptima para su sistema. Para esto el usuario debe ingresar el número de procesos y el quantum de tiempo para el Round Robin.
- **Equidad:** dar a cada proceso en la medida que se pueda una parte justa de la CPU
- **Equilibrio:** mantener en lo que se pueda todas las partes del sistema ocupadas (tener el menor tiempo de CPU muerto)
- Proporcionar una representación realista del proceso de programación y simular con precisión el comportamiento del sistema.
- Debe ser fácil de usar y proporcionar resultados claros y concisos.
- Debe permitir a los usuarios interactuar con el sistema durante la simulación, como pausar o detener la simulación, o cambiar los parámetros del sistema sobre la marcha.
    - Aunque en C no se podría aplicar lo de cambiar parámetros en la marcha, al igual que no se podría pausar la simulación.
    - En el Round Robin se necesita el arrival time, y el burst time pero en el FIFO se necesita una prioridad asimismo en el SJF la prioridad es el tiempo menor de ejecución.
- El simulador debe proporcionar una retroalimentación visual al usuario.
    - En este caso no se puede utilizar un GUI gráfico, por ende, se realizará por consola.
- El algoritmo de round robin utiliza un quantum para asignar tiempo de CPU a cada proceso. El tiempo se asigna según los requerimientos del sistema.
    - Como es una simulación, el tiempo de CPU será asignado por el usuario.
- Hay que tener una cola donde se guardan los procesos que ya están listos para ser utilizados, como se sabe, los procesos tienen tres estados que puede ser Ready, Running o Waiting, por lo que hay que llevar registro.
    - Al utilizar Round Robin un proceso puede ser expulsado de forma involuntaria por haber consumido el tiempo que se le asignó, en cuyo caso se pasa al siguiente proceso que esté en la cola y ese proceso se va al final de la cola.
    - Al utilizar FIFO, los procesos deben esperar a otros que tengan mayor prioridad.

    - Al utilizar SJF, la prioridad va a los procesos con menor tiempo de ejecución
- Implementar un context switching, que es el cambio de proceso a proceso. En el cual se altera por cada proceso los datos que se cargan en memoria.
- Obtener el waiting time, que es el tiempo de espera de un algoritmo en la cola donde se colocan los procesos que ya están listos.
- Obtener el turnaround time, que es el tiempo desde que el proceso inicia hasta que es completo.
- El algoritmo que se implementa debe ser diseñado de acuerdo con la estructura de datos y los datos que se van a manejar.
- Asimismo, si se quiere implementar diferentes algoritmos se debe considerar manejo de funciones separadas para que cualquiera que se quiera utilizar no se relacione o interfiera con otro, y solo se llamaría a la función para mejor desempeño.
    - Además, si se quiere implementar otro se haría otra función o si quiere cambiar el algoritmo solo se modifica la función.
- Tomar en cuenta el planificador sin expulsión (se ejecuta un proceso hasta quedar bloqueado o terminar), y con expulsión (un proceso puede dejar de ejecutar sin haber solicitado un cambio)
    - A parte de la cola de los procesos en estado Ready se debe implementar una cola de procesos que estén bloqueados y que el scheduler pase a los procesos de una cola a otra.
    - Manejo de interrupciones: tomar en cuenta cuando un proceso pasa de un estado a otro por una interrupción.

2. **Implementación en C del simulador:** Está en el archivo adjunto llamado Simulador.c

3. **La implementación de tres algoritmos distintos de scheduling:** Se implementaron los algoritmos de Round Robin, First to Come First to be Served, y Shortest Job First

4. **Resultados:** Se corrió la simulación con 50, 100 y 150 procesos, y con un quantum de 5 segundos.

Obtención de Datos:
A continuación se muestran las tablas que resumen los datos obtenidos al correr el simulador, en base a los criterios de evaluación solicitados. Cabe mencionar que las unidades de los criterios de evaluación están en segundos.

Procesos Balanceados (50 CPU/ 50 IO):
50 procesos:

Criterio de Evolución/ Algoritmos	Round Robin	First to Come First to be Served	Shortest Job First
Throughput	0.0020	0.0023	0.0024
Turnarround Time	986.70	836.26	828.92

Average Response Time	18.22	15.34	9.88

100 procesos:

Criterio de Evolución/ Algoritmos	Round Robin	First to Come First to be Served	Shortest Job First
Throughput	0.0014	0.0017	0.0019
Turnarround Time	1421.74	1156.27	1142.43
Average Response Time	116.98	70.82	37.39

150 procesos:

Criterio de Evolución/ Algoritmos	Round Robin	First to Come First to be Served	Shortest Job First
Throughput	0.0008	0.0010	0.0010
Turnarround Time	2382.45	2008.63	1964.29
Average Response Time	420.21	205.14	57.71

Procesos Desbalanceados (90 CPU/ 10 IO):
50 procesos:

Criterio de Evolución/ Algoritmos	Round Robin	First to Come First to be Served	Shortest Job First
Throughput	0.0032	0.0037	0.0038
Turnarround Time	1103.46	1171.42	1166.62
Average Response Time	6.52	1.52	2.06

100 procesos:

Criterio de Evolución/ Algoritmos	Round Robin	First to Come First to be Served	Shortest Job First
Throughput	0.0011	0.0012	0.0013
Turnarround Time	1910.82	1928.68	1920.45

Average Response Time	29.19	11.10	11.36

150 procesos:

Criterio de Evolución/ Algoritmos	Round Robin	First to Come First to be Served	Shortest Job First
Throughput	0.0008	0.0010	0.0011
Turnarround Time	2994.63	2676.74	2661.51
Average Response Time	83.90	55.61	57.67

Procesos Desbalanceados (10 CPU/ 90 IO):
50 procesos:

Criterio de Evolución/ Algoritmos	Round Robin	First to Come First to be Served	Shortest Job First
Throughput	0.0034	0.0041	0.0042
Turnarround Time	498.34	403.28	396.22
Average Response Time	38.16	34.36	24.58

100 procesos:

Criterio de Evolución/ Algoritmos	Round Robin	First to Come First to be Served	Shortest Job First
Throuput	0.0013	0.0016	0.0017
Turnarround Time	1589.11	1247.40	1207.07
Average Response Time	300.37	188.79	51.50

150 procesos:

Criterio de Evolución/ Algoritmos	Round Robin	First to Come First to be Served	Shortest Job First
Throughput	0.0009	0.0010	0.0011
Turnarround Time	2290.42	1775.61	1640.97

Average Response Time	790.64	648.05	97.75

Tabla de Resumen:
A continuación se muestra la tabla que hace un promedio de todo para sacar un promedio general de cada criterio de evaluación.

General:

Criterio de Evolución/ Algoritmos	Round Robin	First to Come First to be Served	Shortest Job First
Throughput	0.000166	0.000196	0.000206
Turnarround Time	1686.41	1467.14	1436.50
Average Response Time	200.47	136.75	43.47

5. Discusión y Limitaciones:

Al analizar los resultados, se concluyó que dependiendo del criterio de evaluación hay un algoritmo que fue el más eficiente, siendo este el Shortest Job First. En cuanto a los 3 criterios de evaluación es el algoritmo que tiene menor tiempo de respuesta y turnaround time, indicando que es el algoritmo que en promedio se demora menos tiempo en ejecutar un proceso y el que menos tiempo de espera tiene para atender a una solicitud de uso de CPU. Además, es que el mayor throughput en promedio tiene, indicando que es el que más procesos ejecuta/completa en un una unidad de tiempo dado.

Algunas de las limitaciones encontradas en la implementación de este simulador fueron:

- El uso del algoritmo de quicksort para ordenar el arreglo de procesos. Utilizar este algoritmo tiene un costo bastante alto de recursos ya que tiene un Big O de n^2. Se recomienda en el futuro hacer el ordenamiento con un algoritmo de sorting que tenga un Big O menos complejo, como por ejemplo el Merge Sort.
- La implementación de las interrupciones no se hizo del todo de manera aleatoria. Dado a que las interrupciones se definieron antes de que se ejecutará el proceso (en su creación). Se implementaron las interrupciones de esta forma en vez de hacerlas aleatoriamente cuando el proceso entra en ejecución, ya que en C es un poco complejo correr varios procesos al mismo tiempo, dificultando la lógica del tiempo para el simulador.

Algunas de las limitaciones de utilizar los algoritmos de scheduling utilizados son:

- **Round Robin:** Se puede generar un aumento en el tiempo de respuesta promedio debido a que se realizan múltiples context switching. Esto ocurre con mayor probabilidad si se define un valor de quantum demasiado pequeño. Sin embargo, si el

tiempo de quantum es demasiado grande, puede haber una falta de equidad en la asignación de tiempo de CPU.

- **Shortest Job First:** Al tener una naturaleza impredecible de la duración de un proceso, si esta no es conocida de antemano, el algoritmo puede tener dificultades para determinar cuál trabajo es el más corto y, por lo tanto, puede no ser eficiente. Además, si todos los trabajos son de duración similar, el algoritmo puede que no sea capaz de hacer una asignación justa de tiempo de CPU.
- **First-Come, First-Served**: Dado a que no se tiene en cuenta la duración del proceso, si un proceso largo se presenta primero, puede tardar mucho tiempo en completarse. Consecuentemente, llevando a que otros procesos más cortos tengan que esperar mucho tiempo para ser atendido. Esto puede resultar en una mala utilización del tiempo de CPU y en un tiempo de respuesta promedio más largo.

Ejercicio 2: En clase, ustedes analizaron el problema clásico de los Filósofos comensales y revisaron su solución propuesta en el libro. Su trabajo consiste en implementar la solución a este problema en tres lenguajes de programación: C, Java y Python. Utilizando hilos y comparar sus soluciones.

1. Introducción

El problema de los filósofos básicamente se trata de realizar la sincronización de procesos. El problema se basa en la idea de que algunos filósofos (en este caso se implementaron 5) se sientan alrededor de una mesa y se alternan entre las acciones: pensar y comer. Cada filósofo tiene un palillo a su izquierda y derecha, por lo que solo pueden comer si tienen disponibles dos palillos.
Conforme se va avanzando en el proceso se descubre uno de los problemas más importantes que es cuando dos filósofos intentan tomar los palillos al mismo tiempo, lo que causa bloqueos y retrasos en el proceso/acción de comer. Para poder solucionar este problema se debe realizar la coordinación de los procesos y asignación adecuada de recursos, de esta forma se tendrá eficiencia en el sistema. A continuación, se presentan 3 implementaciones en distintos lenguajes de programación (Java, Python y C)

2. Java

La implementación en Java utiliza un enfoque similar al de mutex en C, pero utilizando monitores. En este caso, se crearon hilos para un pool de ejecución donde cada hilo representaba el ciclo de vida de cada filósofo. El cual simula pedir cubiertos al monitor, comer, y pensar 3 veces. Simulando las 3 comidas diarias. El monitor se encarga de administrar el acceso a los cubiertos que son un recurso compartido entre los filósofos, así como también ordenarlos en un orden de espera cuando no hay cubiertos disponibles. En estas situaciones se pone en espera a los hilos que no tienen cubiertos disponibles aún y cuando estos se vuelven disponibles se toma como prioridad a los hilos que llevan mayor tiempo de espera.

Ventajas:

- Los monitores son un patrón de diseño que ayuda a manejar sin conflictos el acceso compartido a recursos y se pueden utilizar con mayor facilidad que los mutexes.
- La implementación es fácil de entender, lo que reduce la posibilidad de errores y la necesidad de depuración.
- Java proporciona una biblioteca estándar de alto nivel, lo que facilita la implementación y la escalabilidad del código.

Desventajas:

- Aunque los monitores son menos propensos a problemas de rendimiento que los mutexes, podrían ser un cuello de botella si utilizáramos una gran cantidad de filósofos por ejemplo ya que aunque existan cubiertos libres solo un filósofo tiene acceso a la vez.
- No se utilizan semáforos en esta implementación, lo que podría ayudar a tener acceso a partes del recurso compartido que otro hilo no está utilizando realmente.
- Java no ofrece el mismo control sobre los recursos del sistema que C, lo que puede limitar la capacidad de optimización del código en algunos casos.

3. Python

Se desarrolla la clase Filósofo, la cual tiene un constructor parametrizado, el cual recibe un índice y dos hilos, que representan los palillos (izquierdo y derecho). La solución se basa en la coordinación de los hilos de ejecución de cada filósofo para evitar bloqueos como se mencionó con anterioridad.
Se utiliza la función run() para iniciar los hilos de ejecución y se definen funciones adicionales como palillo() y comer(). La función principal del programa llama a estos métodos para hacer la respectiva creación de objetos Filósofo y los hilos y luego iniciar el proceso de cada hilo con la función start().

Ventajas

- La librería threading facilita mucho la implementación, ya que proporciona una forma eficiente y sencilla de utilizar hilos de ejecución en el programa.
- Mutex ayuda a evitar la posibilidad de que dos filósofos tomen el mismo palillo en ese instante o más conocido como un deadlock.
- Python simplifica muchas cosas y esto también hace que el código sea fácil de leer y entender, además de que utilizar implementación de clases hace que todo esté más ordenado.
- Es una implementación escalable en el punto de que se pueden manejar más o menos filósofos.

Desventajas

- Python por ser un lenguaje de interpretación es más lento que los lenguajes compilados.
- Al intentar implementar un programa sencillo se evitan funciones de sincronización avanzada y la hace propensa a errores.
- Puede haber consumo excesivo de recursos al utilizar el bucle mientras.

En general, la solución es un buen ejemplo de paralelismo, ofrece una solución efectiva y simple al problema y demuestra cómo se puede controlar el acceso a recursos compartidos para evitar deadlocks.

4. **C**

El código muestra la solución para evitar deadlock utilizando de igual forma mutex los cuales se inicializan en el main(), cada filósofo son hilos y se les pasa el ID como argumento, entonces cada hilo ejecuta la función filosofía() la cual tiene un bucle infinito donde se encuentran las acciones de pensar, comer y tomar/liberar palillos. La función pthread_join() espera a que todos los hilos terminen y finalmente pthread_mutex_destroy() se encarga de liberar los mutex

Ventajas

- Mutex asegura que no haya situaciones de deadlock, permite exclusión mutua, lo cual se basa en garantizar que los recursos compartidos no sean corrompidos por hilos concurrentes.
- La solución es fácil de entender
- Como C es un lenguaje de bajo nivel se tiene un buen control sobre los recursos del sistema.
- Las bibliotecas también son recursos muy útiles ya que facilitan de gran manera la implementación

Desventajas

- Mutex también puede llegar a ser propensa a problemas de rendimiento si son sistemas demasiado cargados.
- Por el motivo anterior es que la implementación puede ser difícil de escalar hasta cierto punto, es decir, que puede ser complicado encontrar una buena solución a medida que los recursos e hilos aumentan.
- Falta de abstracciones de alto nivel en C

5. **Discusiones**

En realidad no hay un mejor lenguaje para implementar el problema de los filósofos, ya que cada lenguaje tiene sus propias ventajas y desventajas. Sin embargo, es importante recalcar que algunos lenguajes pueden ser más adecuados que otros y esto depende de las necesidades específicas de los proyectos.

En Java se puede utilizar el mecanismo de bloqueo de objetos synchronized para garantizar la exclusión mutua y coordinar la ejecución de los hilos. Brinda una serie de herramientas de depuración buenas que pueden ayudar a identificar y solucionar problemas en el código concurrente.

Python como se mencionó anteriormente es un lenguaje de programación de alto nivel que cuenta con una sintaxis sencilla, tiene varias bibliotecas para la concurrencia, como la biblioteca threading, a pesar de que puede ser menos eficiente que C o Java en términos de

velocidad de ejecución, su facilidad de uso y la gran cantidad de bibliotecas lo hacen una buena opción para implementaciones rápidas.

Finalmente, C es un lenguaje de bajo nivel que permite un mayor control sobre la memoria y el rendimiento, el mecanismo de exclusión mutua se basa en mutexes y permite una gestión de recursos del sistema, una de las grandes barreras es que puede ser más difícil de programar pero es una buena opción para implementaciones que requieren eficiencia y rendimiento.

Resumen de actividades:

- **Alejandra:** implementación del simulador (ejercicio 1), implementación de los algoritmos de scheduling, corrección de los requerimientos del scheduler, y discusión y limitaciones del scheduler.
- **Camila:** implementación del simulador (ejercicio 1), implementación de las interrupciones del scheduler, generación y análisis de los datos obtenidos del simulador, y discusión y limitaciones del scheduler.
- **Antonio:** implementación del problema de los filósofos en Java, y discusión, ventajas y desventajas de lo implementado en el ejercicio 2.
- **Pamela:** implementación del problema de los filósofos en Python y C, y discusión, ventajas y desventajas de lo implementado en el ejercicio 2.