Resumen de Programación Concurrente en Java.

1. Conceptos Fundamentales

En el contexto de la programación y el manejo de procesos, es esencial comprender algunos conceptos clave que son la base de los sistemas operativos y la programación concurrente:

- **Programa**: Es un conjunto de instrucciones escritas en un lenguaje de programación que no realiza acciones hasta ser ejecutado. Es un elemento estático.
- **Proceso**: Instancia en ejecución de un programa, gestionada por el sistema operativo, con recursos independientes. Cada proceso tiene su propia memoria y espacio de trabajo.
- **Sistema Operativo**: Gestor que crea y administra los procesos, manejando la comunicación y ejecución de cada uno.
- **Servicio y Daemon**: Tipo de proceso en segundo plano, generalmente ejecutando tareas automáticas, como un firewall o antivirus.
- **Hilo (Thread)**: Unidad de ejecución dentro de un proceso. Los hilos comparten memoria y pueden ejecutar tareas concurrentemente, aunque pueden causar conflictos sin manejo adecuado.

Estados de un Proceso

Los procesos pueden cambiar de estado según su interacción con el sistema y el procesador:
- **Nuevo**: El proceso ha sido creado.
- **Listo**: Espera para ejecutarse.
- **En Ejecución**: Utilizando el procesador.
- **Bloqueado**: Espera a que un evento ocurra, como una operación de entrada/salida.
- **Terminado**: Ha finalizado y el sistema libera sus recursos.

2. Programación Concurrente en Java

Java proporciona múltiples herramientas para crear y gestionar procesos concurrentes. Algunas de las clases y métodos fundamentales incluyen:
- **ProcessBuilder y Runtime.exec()**: Permiten ejecutar comandos y procesos del sistema desde Java.
- **Hilos (Threads)**: Los hilos pueden crearse mediante la clase `Thread` o implementando la interfaz `Runnable`. Métodos como `start`, `join` y `sleep` ayudan a controlar su ejecución.
- **Sincronización con synchronized**: Evita que múltiples hilos accedan a recursos compartidos simultáneamente, usando secciones críticas y monitores.
- **Problemas de Sincronización**: Como interferencia entre hilos y consistencia de memoria, que se pueden manejar mediante métodos sincronizados y bloqueos.

3. Modelos y Técnicas Avanzadas de Sincronización

Para problemas complejos de sincronización, se utilizan patrones avanzados como:
- **Modelo Productor-Consumidor**: Permite que un productor genere datos que un consumidor utiliza, con sincronización mediante `wait` y `notifyAll` para evitar conflictos.

- **Locks y ReentrantLock**: Proporcionan un control más detallado sobre los bloqueos, evitando deadlocks (bloqueos mutuos).

4. Manejo de Ejecución con Executors

Java utiliza `Executors` para gestionar la creación y control de hilos de manera optimizada. Principales interfaces y ejemplos:
- **Executor**: Ejecuta tareas básicas.
- **ExecutorService**: Añade métodos de gestión del ciclo de vida de las tareas, permitiendo también devolver valores mediante `Callable` y `Future`.
- **ScheduledExecutorService**: Programación de tareas periódicas o con retraso.

Thread Pools

Los `Thread Pools` permiten reutilizar hilos ya creados para manejar cargas de trabajo de manera eficiente, evitando los costos de crear y destruir hilos repetitivamente. Algunos tipos son:
- **ThreadPoolExecutor**: Para ejecutar tareas independientes.
- **ScheduledThreadPoolExecutor**: Para ejecutar tareas periódicas o con retraso.
- **ForkJoinPool**: Útil para tareas que pueden dividirse en sub-tareas recursivas.

5. Framework Fork-Join

El `Fork-Join` es un framework diseñado para dividir tareas grandes en subtareas y ejecutarlas en paralelo. Esto se logra usando dos métodos clave:
- **fork()**: Divide la tarea y la envía a la cola de ejecución.
- **join()**: Bloquea el hilo hasta que la tarea finaliza, permitiendo que el `ForkJoinPool` reasigne tareas de manera eficiente.

Ejemplo de Búsqueda del Máximo en un Arreglo

Mediante el uso de `RecursiveTask`, podemos encontrar el valor máximo en un arreglo, dividiendo el arreglo en sub-arreglos y procesando en paralelo para mayor eficiencia.

6. Colecciones Concurrentes y Variables Atómicas

Java ofrece colecciones concurrentes en `java.util.concurrent`, como:
- **BlockingQueue**: Cola FIFO que bloquea si está vacía o llena.
- **ConcurrentMap** y **ConcurrentNavigableMap**: Mapas concurrentes para operaciones seguras en multihilo.

Las **Variables Atómicas** como `AtomicInteger` permiten realizar operaciones como `incrementAndGet` sin necesidad de sincronización explícita.