Sincronización y Algoritmos de Scheduling

Autores: Luis Fernando Benavides Villegas – Alex Naranjo Masis
Instituto Tecnológico de Costa Rica – Principios de Sistemas Operativos

Descripción de la Solución

La solución implementa una **línea de ensamblaje concurrente** compuesta por tres estaciones de trabajo. Las estaciones de *Corte*, *Ensamblaje* y *Empaque* procesan productos de manera secuencial mediante comunicación entre hilos independientes y colas compartidas sincronizadas. Cada estación puede configurarse para ejecutar los algoritmos de planificación **First-Come, First-Served (FCFS)** o **Round Robin (RR)**, con quantum configurable desde la terminal.

La simulación está desarrollada en **Rust**, aprovechando las abstracciones de concurrencia seguras del lenguaje. El programa principal (`main.rs`) hace lo siguiente:

1. Un generador de productos que encola 10 elementos con tiempos de llegada simulados (offsets en milisegundos).
2. Tres hilos independientes, cada uno representando una estación de trabajo.
3. Cada estación lee de una cola de entrada bloqueante, simula su tiempo de procesamiento mediante `sleep()`, y coloca el producto en la cola de salida correspondiente.

La estructura `Product` guarda métricas por estación: tiempos de entrada, salida, duración total y espera total, y al finalizar, un recolector extrae los productos terminados y genera un informe tabulado con métricas individuales y promedios globales.

El flujo general de ejecución es:

```
Generador → [Cola E1] → Estación 1 → [Cola E2] → Estación 2 → [Cola E3] → Estación 3 → [Cola
Done] → Recolector
```

Cada estación es un hilo separado que ejecuta indefinidamente, procesando un producto a la vez y garantizando exclusión mutua a nivel de acceso a los datos. El algoritmo de planificación determina cómo se gestiona la cola interna de cada estación:

- **First-Come, First-Served:** Procesa cada producto completamente antes de tomar el siguiente.
- **Round Robin:** Procesa por secciones de tiempo (`quantum`) y reencola los productos que no han completado su tiempo total de trabajo.

Ejemplo de Ejecución

Con el comando:

```
cargo run -- fcfs 140 rr 220 80 fcfs 120
```

Logramos esta configuración de las estaciones:

```
Tiempos de procesamiento por estación:
 - Corte      (#1, FCFS)            → 140 ms
 - Ensamblaje (#2, RR, q=80)        → 220 ms
 - Empaque    (#3, FCFS)            → 120 ms
```

Y nos retornará este informe:

ID	Arr(ms)	E1_in	E1_out	Wait1	E2_in	E2_out	Wait2	E3_in	E3_out	Wait3	Turn(ms)	WaitTot
0	0	0	141	1	141	445	84	446	566	1	566	86
1	0	142	282	142	303	668	165	669	789	1	789	309
2	50	283	423	233	526	1051	407	1053	1173	1	1123	643
3	101	425	565	324	668	1273	487	1274	1395	1	1294	814
4	202	566	707	365	829	1577	650	1577	1697	0	1495	1015
5	304	708	848	404	1052	1961	892	1962	2082	1	1778	1298
6	455	849	990	395	1274	2102	891	2103	2224	1	1769	1289
7	656	991	1131	335	1355	2244	892	2245	2365	0	1709	1229
8	907	1132	1272	225	1577	2305	812	2366	2487	61	1580	1100
9	1207	1273	1413	66	1738	2367	733	2488	2608	121	1401	921
AVG	—	—	—	**249.00**	—	—	**601.30**	—	—	**18.80**	**1350.40**	**870.40**

Justificación Técnica

La solución fue implementada íntegramente en Rust, escogido por su modelo de concurrencia libre de data races y por ofrecer control de bajo nivel con verificación de seguridad en tiempo de compilación. Esto lo convierte en una alternativa moderna frente a C o Go, especialmente en un entorno educativo donde se desea experimentar con mecanismos de sincronización reales (mutexes, colas bloqueantes y condicionales) sin incurrir en errores típicos de punteros o condiciones de carrera.

Diseño General y Estructura Modular

El proyecto se divide en tres módulos principales:

- `estaciones.rs`: Define estructuras compartidas, colas sincronizadas (ProdQueue) y funciones auxiliares de tiempo.
- `funciones.rs`: Implementa los algoritmos de planificación FCFS y Round Robin.
- `main.rs`: Parsea argumentos, coordina la creación de hilos, colas y generación de productos, además de recolectar y reportar métricas.

Esta organización modular permite independencia funcional, facilita pruebas unitarias y promueve la reutilización del código.

1. Comunicación y Exclusión Mutua (`estaciones.rs`)

El módulo `estaciones.rs` implementa el núcleo del mecanismo de intercomunicación entre procesos mediante la estructura `ProdQueue`. Esta cola usa un `Arc<Mutex<Inner>>` junto con variables de condición (`Condvar`) para sincronizar productores y consumidores.

- *Arc (Atomic Reference Counted): Permite compartir de forma segura la misma estructura entre varios hilos.*
- *Mutex (Mutual Exclusion): Garantiza que solo un hilo acceda o modifique los datos de la cola a la vez.*
- *Condvar (Condition Variable): Suspende un hilo hasta que se cumpla una condición concreta*

Mecanismo de Bloqueo

```
// Encola un elemento. Si la cola está llena, se bloquea hasta que haya espacio.
pub fn push(&self, item: SharedProduct) {
    let mut g = self.inner.lock().unwrap();     // Toma el lock del estado interno.
    while g.buf.len() == self.capacity {
        g = self.not_full.wait(g).unwrap();     // Mientras esté llena, espera a `not_full`.
    }
    g.buf.push_back(item);
    self.not_empty.notify_one();                // Despierta a un consumidor que esté esperando a
`not_empty`.
}

// Desencola un elemento. Si la cola está vacía, se bloquea hasta que haya ítems.
pub fn pop(&self) -> SharedProduct {
    let mut g = self.inner.lock().unwrap();     // Toma el lock del estado interno.
    while g.buf.is_empty() {
        g = self.not_empty.wait(g).unwrap();    // Mientras esté vacía, espera a `not_empty`
    }
    let item = g.buf.pop_front().expect("checked non-empty");
    self.not_full.notify_one();                 // Despierta a un productor que esté esperando a
`not_full`.
    item
}
```

Aquí, `push()` se bloquea si la cola está llena, y `pop()` hace lo propio si está vacía. Esto simula fielmente la comunicación bloqueante entre estaciones, evitando busy waiting y garantizando que solo un hilo manipule el buffer a la vez. Las operaciones se despiertan mutuamente mediante las condiciones `not_full` y `not_empty`, lo cual permite flujo continuo y evita interbloqueos.

Cada producto se representa como un `Arc<Mutex<Product>>` (`SharedProduct`), que permite compartirlo entre hilos con acceso controlado a sus métricas internas.

2. Planificación de Estaciones (`funciones.rs`)

El módulo `funciones.rs` contiene la lógica de los algoritmos **First-Come, First-Served** y **Round Robin**, ambos ejecutados en bucles infinitos dentro de hilos dedicados a cada estación.

FCFS: Procesamiento Secuencial

```
let sp = in_q.pop();

sleep_ms(work_time_ms);

q_out.push(sp.clone());
```

En este algoritmo, cada producto es atendido completamente antes de pasar al siguiente. El hilo de la estación se mantiene bloqueado hasta recibir un nuevo producto, asegurando procesamiento exclusivo por estación y sin interferencias concurrentes.

RR: Procesamiento por Quantum

```
let sp = in_q.pop();
let slice = p.remaining_rr.max(0).min(quantum_ms);

sleep_ms(slice);

p.remaining_rr -= slice;

if p.remaining_rr <= 0 {
    q_out.push(sp.clone());
} else {
    in_q.push(sp.clone());
}
```

Aquí cada producto se procesa por un *"slice"* de tiempo (`quantum_ms`) y, si no termina, se reencola en la misma cola (`in_q.push()`) para la próxima ronda. Este comportamiento emula la planificación con interrupciones temporales apropiativas, simulando el comportamiento de una CPU multitarea. El campo `remaining_rr` dentro de `Product` mantiene el estado entre reencolados, y los tiempos de entrada/salida se actualizan bajo Mutex, garantizando consistencia.

Ambas funciones se benefician de la seguridad de Rust, por lo que si un hilo intenta acceder simultáneamente a un `Product` sin lock, el compilador lo detecta como error.

3. Coordinación Global y Medición (`main.rs`)

El módulo `main.rs` actúa como el planificador maestro. Primero define las configuraciones de estación (`StationCfg`) con tipo (`FCFS` o `RR`) y parámetros (`work_ms`, `q`), y luego inicializa colas `ProdQueue` entre estaciones y el reloj global con base al que se toma el tiempo en todas las estaciones:

Generador de Productos

El generador es un hilo que simula tiempos de llegada (`arrival_ms`) y crea productos compartidos (`SharedProduct`). Cada producto es encolado en la primera estación de acuerdo con un offset temporal programado, simulando llegadas asincrónicas:

```
sleep_ms(offset_ms);
q_e1_in.push(sp);
```

Ejecución Concurrente de Estaciones

Cada estación se lanza en su propio hilo con `thread::spawn`, recibiendo clones de colas (`Arc::clone`) y referencias a las funciones `estacion_fcfs` o `estacion_round_robin` según el tipo de planificación.

Recolector e Informe Final

El recolector (`q_done.pop()`) centraliza las métricas una vez que los productos terminan. Calcula los tiempos de turnaround, espera total y espera por estación. Los resultados se muestran tabulados con promedios.

4. Ventajas Técnicas y Robustez

- **Sincronización eficiente:** El uso de `Condvar` evita el desperdicio de CPU (a diferencia de un busy loop).
- **Evita interbloqueos:** El diseño circular con una única cola entre estaciones mantiene flujo unidireccional.
- **Seguro ante errores de concurrencia:** `Arc<Mutex>` garantiza acceso exclusivo y seguro.
- **Escalable:** Basta agregar más estaciones con nuevas colas y hilos sin alterar la lógica de sincronización.

Comparación entre Algoritmos

Para evaluar el rendimiento de los algoritmos de planificación implementados, se realizaron pruebas equivalentes utilizando la misma configuración de llegada y tiempo de servicio. En ambos casos, se simularon dos productos con tiempos de llegada simultáneos y una estación de trabajo con tiempo total de procesamiento de 220 ms. La diferencia radica en la política de planificación utilizada: **FCFS** y **RR** con un quantum de 80 ms.

First Come First Serve

En este algoritmo, los productos son atendidos estrictamente en el orden en que llegan a la cola, y cada estación procesa un producto completamente antes de atender el siguiente. No existe interrupción ni reencolado, por lo que el comportamiento es determinista y secuencial.

El comando de ejecución fue:

```
cargo run -- fcfs 120 fcfs 220 fcfs 100
```

```
Tiempos de procesamiento por estación:
 - Corte      (#1, FCFS)            → 120 ms
 - Ensamblaje (#2, FCFS)            → 220 ms
 - Empaque    (#3, FCFS)            → 100 ms
```

ID	Arr(ms)	E1_in	E1_out	Wait1	E2_in	E2_out	Wait2	E3_in	E3_out	Wait3	Turn(ms)	WaitTot
0	0	0	120	0	121	342	1	342	443	1	443	3
1	0	121	241	121	342	562	100	564	664	1	664	224
2	50	242	362	192	563	784	201	785	885	1	835	395
3	101	363	483	262	784	1005	302	1005	1105	0	1004	564
4	202	483	604	282	1005	1225	401	1226	1326	0	1124	684
5	302	604	724	302	1226	1446	501	1447	1548	1	1246	806
6	453	725	846	273	1447	1667	601	1668	1768	0	1315	875
7	654	847	967	193	1668	1888	701	1889	1990	1	1336	896
8	905	968	1088	63	1889	2109	800	2111	2211	1	1306	866
9	1205	1206	1326	1	2111	2331	784	2331	2432	1	1227	787
AVG	—	—	—	**168.90**	—	—	**439.20**	—	—	**0.70**	**1050.00**	**610.00**

Análisis

El algoritmo FCFS muestra un comportamiento estable y sin tantos cambios de contexto. Los productos son procesados de forma ordenada, y el sistema mantiene un flujo continuo sin interrupciones. Sin embargo, el tiempo promedio de espera (610 ms) evidencia el efecto de convoy, ya que los productos que llegan posteriormente deben esperar a que los anteriores finalicen completamente su paso por cada estación.

Ventajas

- Simplicidad y predictibilidad.
- Adecuado cuando las tareas tienen tiempos de servicio similares.

Desventajas

- Ineficiente para cargas mixtas o heterogéneas (en este caso, todas las estaciones tienen una duración fija, pero si los productos tuvieran su propio burst cambiarían las cosas).
- Puede generar largos tiempos de espera acumulativos y starving.

Round Robin (Quantum = 80)

El algoritmo Round Robin introduce preempción temporal, cada producto recibe una fracción de tiempo (quantum) antes de ser interrumpido y reencolado. Esto produce mayor alternancia entre productos, mejorando la equidad, pero con un costo adicional por los cambios de contexto y la fragmentación de servicio.

El comando de ejecución fue:

```
cargo run -- rr 120 80 rr 220 80 rr 100 80
```

```
Tiempos de procesamiento por estación:
 - Corte      (#1, RR, q=80)        → 120 ms
 - Ensamblaje (#2, RR, q=80)        → 220 ms
 - Empaque    (#3, RR, q=80)        → 100 ms
```

ID	Arr(ms)	E1_in	E1_out	Wait1	E2_in	E2_out	Wait2	E3_in	E3_out	Wait3	Turn(ms)	WaitTot
0	0	0	282	162	282	585	83	586	687	2	687	247
1	0	80	404	284	444	889	265	890	992	2	992	552
2	50	161	525	355	666	1354	608	1354	1456	2	1406	966
3	101	282	647	426	747	1739	872	1740	1842	2	1741	1301
4	202	404	768	446	970	2043	1054	2043	2145	2	1943	1503
5	303	526	810	387	1051	2184	1154	2185	2368	83	2065	1625
6	454	647	931	357	1212	2245	1093	2266	2390	44	1936	1496
7	655	810	972	197	1354	2387	1194	2390	2572	84	1917	1477
8	906	973	1094	68	1517	2448	1134	2470	2673	124	1767	1327
9	1206	1207	1328	2	1820	2509	961	2572	2694	84	1488	1048
AVG	—	—	—	**268.40**	—	—	**841.80**	—	—	**42.90**	**1594.20**	**1154.20**

Análisis

El algoritmo RR incrementa el tiempo promedio de espera total a 1154 ms (casi el doble que FCFS) y el turnaround promedio a 1594 ms, producto de los múltiples reencolados e interrupciones. Sin embargo, permite que todos los productos avancen de manera progresiva, reduciendo el tiempo de respuesta percibido y evitando el bloqueo de tareas cortas por tareas largas.

Ventajas

- Mayor equidad en la asignación del recurso.
- Mejora la responsividad global del sistema.

Desventajas

- Overhead adicional por cambios de contexto.
- Turnaround promedio más alto que en FCFS.

Comparativa Final

Métrica	FCFS (ms)	RR (ms)	Diferencia
Turnaround promedio	1050.0	1594.2	544.2
Espera total promedio	610.0	1154.2	544.2
Wait promedio E1	168.9	268.4	99.5
Wait promedio E2	439.2	841.8	402.6
Wait promedio E3	0.7	42.9	42.2

Los resultados demuestran que FCFS es más eficiente en términos de throughput y tiempo promedio, pero sacrifica equidad y tiempo de respuesta inicial. Por el contrario, Round Robin mejora la equidad y simula un entorno más realista para sistemas multitarea, a costa de un mayor overhead y tiempos de espera totales. En un contexto industrial o de simulación de línea de ensamblaje, FCFS es más adecuado cuando se busca maximizar productividad y predictibilidad en estaciones que tienen una duración de procesado fija, mientras que RR resulta útil cuando se prioriza la distribución justa del recurso o la representación de un comportamiento preemptivo tipo CPU.

Criterio	FCFS (First-Come, First-Served)	RR (Round Robin, q=80 ms)
Política de planificación	No apropiativa. Atiende cada producto completo antes de pasar al siguiente.	Apropiativa. Interrumpe el procesamiento tras un *quantum* y reencola el producto.
Orden de atención	Estrictamente según orden de llegada (determinista).	Todos los productos avanzan progresivamente (cíclico y equitativo).
Control de flujo	Una cola FIFO por estación.	Requiere gestión de reencolado y conteo de tiempo restante.
Uso de CPU / recursos	Menor overhead, sin interrupciones ni cambios de contexto.	Mayor overhead por interrupciones y operaciones de reencolado.
Tiempo de espera promedio	Más bajo. Menos alternancia, pero más acumulación para tareas largas.	Más alto. Aumenta por fragmentación temporal y sobrecarga de contexto.
Turnaround promedio	Más bajo; flujo lineal y predecible.	Más alto; cada producto tarda más en completarse por reencolado frecuente.
Equidad entre productos	Baja. Las primeras tareas monopolizan la estación.	Alta. Cada producto obtiene oportunidad de avance en cada ciclo.

Criterio	FCFS (First-Come, First-Served)	RR (Round Robin, q=80 ms)
Efecto "convoy"	Presente. Los productos cortos esperan a los largos.	Eliminado. Los productos cortos progresan junto a los largos.
Adecuado para	Sistemas con cargas homogéneas y operaciones determinísticas como las líneas de ensamblaje fijas.	Sistemas con procesos con tiempos de burst distintos a donde la equidad es más importante.
Complejidad de implementación	Baja: basta una cola FIFO y `sleep()` por producto.	Media: requiere control de *quantum*, reencolado y actualización de tiempo restante.
Problemas comunes	Starvation de productos que llegan tarde si el flujo es continuo.	Overhead excesivo si el *quantum* es demasiado pequeño; si es muy grande funciona como fcfs.
Resumen general	Mayor rendimiento promedio y predictibilidad, pero menor justicia.	Mayor equidad, pero con más espera total y turnaround.