# Explicación Técnica – Simulador del Ciclo de Vida de Procesos

## 1. Introducción

El presente documento describe la arquitectura y funcionamiento interno del Simulador de Ciclo de Vida de Procesos desarrollado en React + Vite, estas tecnologías fueron seleccionadas porque React es perfecto para construir interfaces de usuario complejas y dinámicas como la de este simulador y por su parte, Vite dio un entorno de desarrollo rápido para trabajar con una compilación optimizada para la versión final.

## 2. Arquitectura General

El simulador está diseñado bajo una arquitectura React Component + Services, lo que permite separar la interfaz de usuario de la lógica interna de la simulación. A continuación, se describe cada parte y su función dentro del sistema:

### 2.1. AppShell (Contenedor Principal)

Organiza la estructura general de la aplicación: encabezado, diagrama de estados, barra de control y panel de información.Define el layout y la navegación básica de la interfaz.

### 2.2. **Contexto Global (SimulationContext**)

- Centraliza el estado de la simulación (procesos, modo de ejecución, velocidad).
- Permite que todos los componentes accedan y modifiquen el estado sin necesidad de pasar datos manualmente entre ellos.
- Sincroniza la interfaz con el motor de simulación.

### 2.3. Hook useSimulation:

- Implementa la lógica principal de gestión de procesos usando useReducer.
- Maneja acciones como crear, actualizar y terminar procesos.
- Expone funciones para interactuar con los procesos desde cualquier componente.

### 2.4. Motor de Simulación (engine.js)

- Controla el avance automático o manual de los procesos.
- Aplica las reglas de transición y gestiona los temporizadores de ejecución.
- Notifica cambios al contexto global para mantener la interfaz sincronizada.

### 2.5. FSM (fsm.js)

- Define los estados posibles de los procesos y sus transiciones válidas.
- Valida los cambios de estado y registra el historial con sus respectivos timestamps.

### 2.6.Diagrama de Estados

- Representa gráficamente el ciclo de vida de cada proceso.
- Muestra nodos (estados) y aristas (transiciones) en tiempo real.
- Facilita la comprensión del flujo de los procesos durante la simulación.

## 3. Modelo de Datos del Proceso

Cada proceso se modela como un objeto plano que cuenta con la siguiente estructura:

| Campo | Tipo | Descripción |
|---|---|---|
| pid | String | Identificador único |
| state | Enum | Estado actual (New, Ready, Running…) |
| priority | Number | Prioridad (0–9) |
| pc | Number | Contador de programa simulado |
| cpuRegisters | Object | Snapshot de registros |

| | | |
|---|---|---|
| syscalls | Array | Llamadas al sistema registradas |
| history | Array | Historial de transiciones {from,to,timestamp,cause} |
| createdAt | Date | Fecha de creación |
| stateEnteredAt | Date | Fecha en la que ingresó al estado actual |

# 4. Máquina de Estados Finita (FSM)

El archivo fsm.js implementa las **reglas de transición**:

**Estados posibles:**

- **New:** Proceso recién creado.
- **Ready:** Proceso listo para ser ejecutado.
- **Running**: Proceso en ejecución.
- **Waiting:** Proceso esperando por una operación de E/S.
- **Terminated:** Proceso finalizado.

**Transiciones válidas:**

- New → Ready
- Ready → Running
- Running → Ready (preempt)
- Running → Waiting (request I/O)
- Running → Terminated
- Waiting → Ready (I/O complete)

**Función principal: transition(process, toState, cause)**

La función transition se encarga de gestionar los cambios de estado de un proceso.

Comportamiento:

**Validación de transición:** Verifica si el cambio hacia toState está permitido desde el estado actual del proceso.

**Si la transición es válida:**Registra el cambio en el historial del proceso (history[]) y actualiza el campo stateEnteredAt, lo que permite medir el tiempo de permanencia en cada estado.

**Si la transición es inválida:**Genera un Error, indicando que la transición solicitada no está permitida.

# 5. Motor de Simulación (engine.js)

El motor es responsable de **avanzar procesos automáticamente**:

- **Modos:** manual y auto.
- **Velocidad:** intervalo de ticks configurable (speed en ms).

**Flujo:**

1. Inicialización
    - El motor se configura en modo manual o automático y se establece la velocidad (speed) en milisegundos.
2. Modo Automático
    - Se utiliza setInterval(step, speed) para ejecutar el ciclo de simulación periódicamente.
    - En cada ciclo (step), el motor recorre todos los procesos activos (excepto los que están en estado Terminated).
    - Para cada proceso, verifica el estado actual y el tiempo transcurrido en ese estado.
    - Si el tiempo en el estado supera el valor definido en DURATIONS, se realiza la transición correspondiente:
        i. New → Ready
        ii. Ready → Running (puede considerar prioridad)
        iii. Running → Terminated o Running → Waiting (según probabilidad)
        iv. Waiting → Ready (cuando termina la operación de I/O)
    - Los procesos en estado Terminated son ignorados y no vuelven a transicionar.
3. Modo Manual
    - El usuario controla las transiciones de los procesos mediante la interfaz.
    - No hay avance automático ni transiciones por inactividad.
    - Notificación de cambios

Cada vez que ocurre una transición, la función onUpdate notifica los cambios a React, provocando el re-renderizado de la interfaz y actualizando la visualización de los procesos.

Este flujo asegura que los procesos avancen correctamente entre estados, respetando los tiempos definidos y las reglas de transición, y que la interfaz se mantenga sincronizada con el estado interno del motor.

-

# 7. Hook useSimulation y Contexto

El simulador utiliza un mecanismo interno llamado useSimulation junto con un contexto global (SimulationContext) para manejar de forma centralizada todo el estado de la simulación de procesos.

**¿Cómo funciona?**

- Se encarga de mantener y actualizar la información de los procesos (por ejemplo: lista de procesos, estado, velocidad y modo de simulación).
- Los cambios que ocurren en la simulación se reflejan automáticamente en toda la interfaz, sin necesidad de que el usuario actualice nada manualmente.
- Gracias a este sistema, los diferentes módulos del simulador (diagrama, controles, panel de información, etc.) siempre muestran la información actualizada y coherente.

**¿Por qué es útil?**

- Permite que el simulador sea más claro, ordenado y fácil de usar.
- Hace posible que nuevas funciones se integren en el futuro sin alterar la experiencia del usuario.
- Mantiene la interfaz re**activa**, es decir, cada cambio se ve reflejado de inmediato.

**Beneficio para el usuario**

El usuario obtiene una experiencia más intuitiva y fluida, ya que cada acción (crear procesos, detenerlos, cambiar la velocidad, etc.) se refleja en tiempo real en toda la aplicación, evitando inconsistencias y garantizando un manejo eficiente de la simulación.

# 8. Interfaz de Usuario

La interfaz del simulador de procesos está diseñada para ser **intuitiva, visual y educativa**, permitiendo comprender de manera interactiva la gestión de procesos en sistemas operativos.

**1. Encabezado y estructura general**

- **AppShell:** Componente raíz que organiza la aplicación en secciones principales.
- Incluye un encabezado con el título del simulador y accesos a opciones adicionales, como la visualización de detalles técnicos.

**2. Diagrama de estados**

- **Visualización gráfica:** Utiliza la librería `@xyflow/react` para representar los estados del proceso (*New, Ready, Running, Waiting, Terminated*) como nodos conectados por transiciones válidas.
- **Nodos de proceso:** Cada proceso activo aparece como un nodo que se desplaza entre estados de acuerdo con las reglas del motor. Las animaciones (`motion`) facilitan la identificación de los cambios.
- **Colores y posiciones:** Cada estado posee un color distintivo y una ubicación fija en el diagrama, lo que permite reconocerlos de inmediato.

**3. Barra de control**

- **Botones principales:**
    - *Crear:* agrega nuevos procesos.
    - *Exportar CSV:* descarga el historial de procesos para análisis externo.
- **Modos de simulación:**
    - *Manual:* el usuario controla las transiciones.
    - *Automático:* el motor ejecuta las transiciones de forma continua.
    - *Pausa:* detiene temporalmente la ejecución.
- **Control de velocidad:** un *slider* ajusta la rapidez de los ciclos automáticos.

- **Estados de los botones:** algunos botones se desactivan según el modo para evitar acciones no válidas (ejemplo: en automático no es posible avanzar manualmente).

**4. Panel de información**

- **Lista de procesos:** muestra todos los procesos activos con su *PID*, estado actual y momento de entrada al estado.
- **Historial de transiciones:** cada proceso cuenta con un registro detallado de sus cambios de estado, incluyendo *timestamps* y causas.
- **Métricas:** ofrece datos como tiempo en cada estado y tiempo total de vida del proceso.

**5. Menú contextual de procesos**

- **Acciones según estado:** al seleccionar un proceso, se despliega un menú con las operaciones disponibles (admitir, asignar CPU, solicitar I/O, terminar, etc.).
- **Validación de acciones:** solo se muestran opciones válidas según el estado y el modo de simulación activo.

**6. Experiencia visual y de usuario**

- **Animaciones:** los cambios de estado están animados para resaltar la dinámica del ciclo de vida de los procesos.
- **Notificaciones y sonidos:** opcionalmente se reproducen sonidos en transiciones clave.
- **Interacción directa:** el usuario puede alternar entre modos de simulación y observar los efectos en tiempo real.

**7. Exportación y reportes**

- **Exportar CSV:** permite descargar el historial de procesos con sus transiciones y marcas de tiempo.

# 9. Dependencias y Justificación

El proyecto utiliza diversas librerías y herramientas que facilitan la construcción de la interfaz, la simulación de procesos y la generación de reportes. A continuación, se presentan las

**dependencias** y **devDependencies** incluidas en el archivo `package.json`, junto con su propósito en el sistema:

**Dependencias**

- **@xyflow/react**
  Permite la visualización de diagramas de flujo y grafos, fundamentales para representar gráficamente los estados y transiciones de los procesos.

- **date-fns**
  Facilita la manipulación y el formateo de fechas y horas, lo cual es clave para mostrar *timestamps* y calcular métricas de tiempo en la simulación.

- **howler**
  Maneja la reproducción de sonidos en la interfaz, por ejemplo, para notificaciones o efectos al producirse eventos en el simulador.

- **motion**
  Proporciona animaciones fluidas en la interfaz, como las transiciones visuales de nodos y procesos.

- **papaparse**
  Convierte datos a formato **CSV**, permitiendo la exportación de reportes y el análisis externo de la simulación.

- **react**
  Biblioteca principal para construir la interfaz de usuario del simulador.

- **react-dom**
  Encargada del renderizado de los componentes React en el DOM del navegador.

- **react-force-graph-2d**
  Ofrece visualización avanzada de grafos en 2D, útil para representar relaciones más complejas entre procesos.

- **update**
  Utilidad para la manipulación de objetos. Se recomienda verificar su uso real en el código, ya que no es muy común.

- **uuid**
  Genera identificadores únicos, empleados para diferenciar procesos y elementos gráficos dentro de la simulación.

**DevDependencies**

- **@eslint/js**
  Conjunto de reglas de ESLint para garantizar buenas prácticas en el código JavaScript.

- **@testing-library/jest-dom**
  Extensiones que facilitan las pruebas en el DOM durante el desarrollo.

- **@testing-library/react**
  Herramientas para realizar pruebas sobre componentes React.

- **@vitejs/plugin-react**
  Plugin que integra React con Vite, mejorando la velocidad de desarrollo y la experiencia de compilación.

- **eslint**
  Analizador de código estático para mantener calidad, estilo y consistencia en el proyecto.

- **eslint-plugin-react-hooks**
  Reglas específicas para garantizar el uso correcto de hooks en React.

- **eslint-plugin-react-refresh**
  Soporte para *Fast Refresh* en React, lo que permite ver cambios en tiempo real durante el desarrollo.

- **globals**
  Define variables globales para la correcta configuración de ESLint.

- **vite**
  Bundler y servidor de desarrollo rápido, utilizado para construir y ejecutar el simulador.

- **vitest**
  Framework de testing que permite ejecutar pruebas unitarias y de integración en proyectos basados en Vite.

# 10. Algoritmos de Cálculo de Tiempos

El simulador lleva un control preciso de las transiciones de cada proceso para calcular su permanencia en los distintos estados.

**Registro de transiciones:**

Cada vez que un proceso cambia de estado, se almacena en su historial (history[]) un objeto con la siguiente información:

- from: estado de origen.
- to: estado de destino.
- timestamp: momento exacto de entrada al nuevo estado.
- cause: motivo de la transición (ejemplo: manual, auto, timeout, prioridad).
- Datos adicionales: valores como pc, cpuRegisters y priority.

**Cálculo de duración real:**

La duración en un estado se obtiene comparando el timestamp de entrada con el timestamp de salida (cuando ocurre la siguiente transición).

Ejemplo:

Si un proceso pasa de READY → RUNNING a las 10:00:05 y luego de RUNNING → WAITING a las 10:00:08, la duración en el estado RUNNING es de 3 segundos (3000 ms).

# 11. Manejo de Errores

- Validación de PID inexistente:En los métodos definidos en useSimulation.js (por ejemplo: admit, assignCPU, entre otros), el sistema verifica que el proceso con el PID especificado exista antes de ejecutar la acción.Si el PID no existe, se genera un Error, evitando operaciones inválidas sobre procesos inexistentes.

```
if (!proc) throw new Error('PID no encontrado')
```

- Transiciones inválidas: En el archivo fsm.js, la función transition comprueba si la transición solicitada entre estados es válida.En caso de no estar permitida, se lanza un Error, el cual puede ser capturado y mostrado tanto en la consola como en la interfaz de usuario (UI).

```
if (!allowed.includes(toState)) {
  throw new Error(`Transición inválida: ${fromState} → ${toState} (PID ${process.pid})`);
}
```

- Procesos en estado Terminated:Los procesos que alcanzan el estado Terminated no poseen transiciones válidas.El motor de simulación los descarta automáticamente, impidiendo que se intenten avanzar y evitando así posibles ciclos infinitos.

# 12. Conclusiones y Futuras Mejoras

El simulador de procesos que desarrollamos es una herramienta muy útil para aprender y entender cómo funcionan los sistemas operativos por dentro. Nos permitió ver de manera interactiva cómo cambian los estados de los procesos, cómo se manejan las transiciones y cómo el motor de simulación controla todo el ciclo de vida. Esto hace que conceptos que suelen ser complejos se entiendan más fácil y motiva a experimentar directamente con los procesos.

Además, el proyecto deja una base sobre la cual se pueden seguir agregando cosas nuevas. Algunas mejoras que pensamos podrían implementarse son:

- **Algoritmos de planificación:** incluir algoritmos como *Round Robin*, *Shortest Job First (SJF)* o planificación por prioridades. Con esto sería posible comparar cómo funcionan distintas formas de asignar la CPU y analizar cuál es más eficiente.

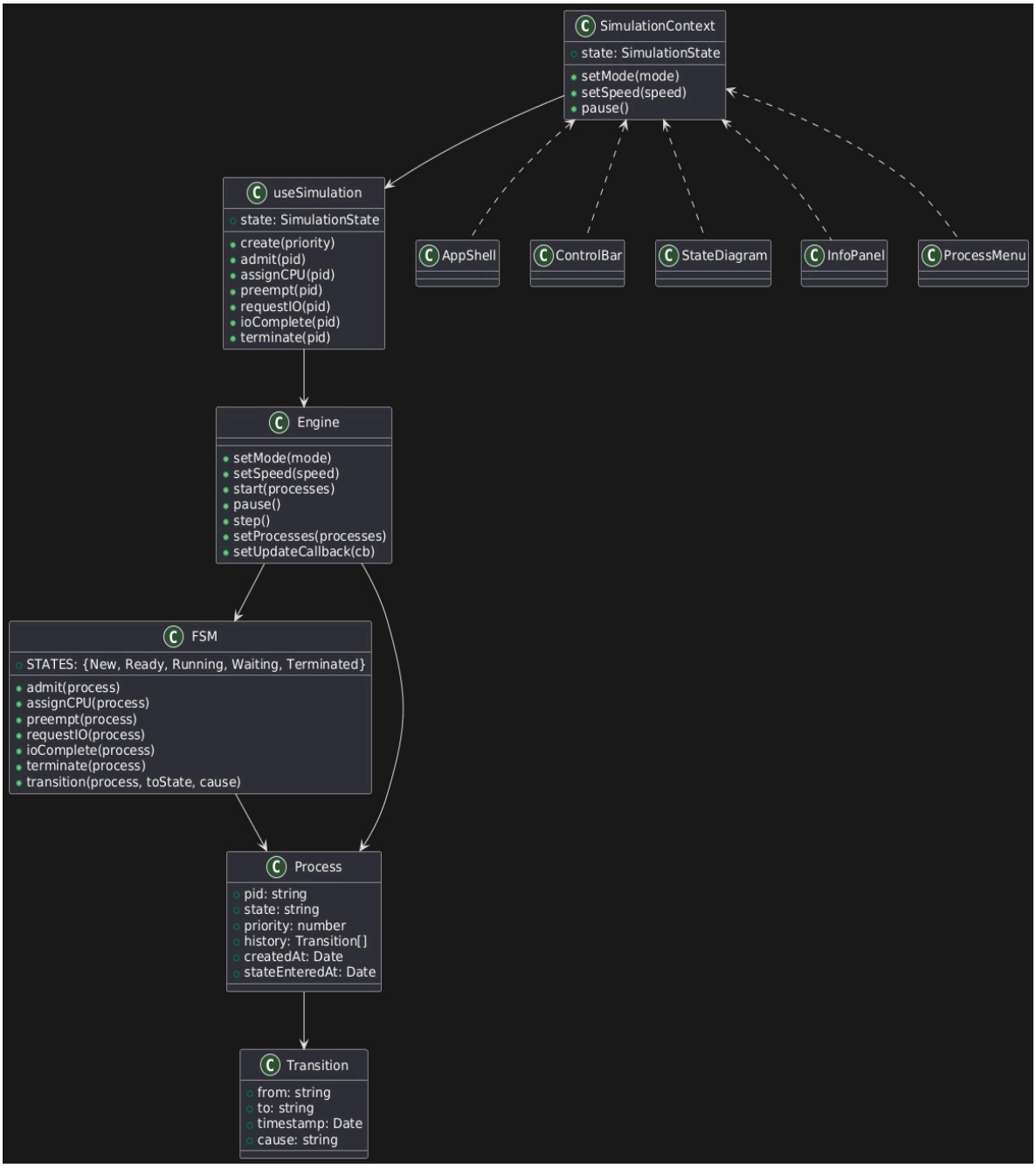
- **Persistencia de datos:** guardar la simulación en *LocalStorage* para poder continuarla en otra sesión y tener un historial de ejecuciones.

- **Métricas avanzadas:** añadir cálculos como el tiempo de respuesta, turnaround y throughput para evaluar el rendimiento y hacer comparaciones entre algoritmos y configuraciones.

En conclusión, el simulador cumple con el objetivo principal de mostrar cómo se gestionan los procesos en un sistema operativo, pero también abre la posibilidad de seguir creciendo hasta convertirse en una herramienta más completa y práctica para el aprendizaje, la investigación y la experimentación.

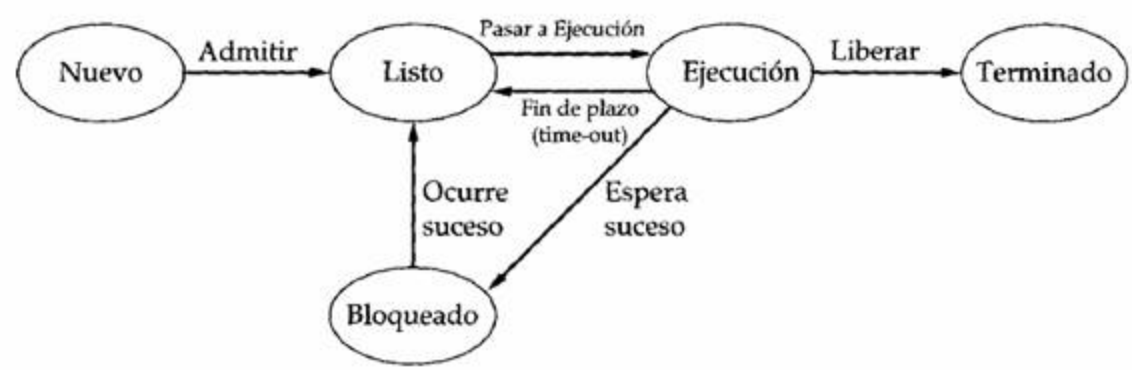
# 13. Anexos

- Diagrama UML
  En el siguiente diagrama UML de clases se representa la arquitectura general del simulador de procesos. Se muestran los principales componentes (contexto, motor, máquina de estados, procesos y transiciones) junto con sus atributos y métodos. Además, se visualizan las relaciones entre las distintas partes del sistema, lo cual permite comprender cómo interactúan para dar soporte a la simulación.




- **Diagrama de Estados**

  **El siguiente diagrama de estados representa las diferentes etapas por las que pasa un proceso dentro del simulador, así como las posibles transiciones entre ellas. Permite visualizar de manera clara el ciclo de vida de un proceso, desde su creación hasta su finalización.**




- **Flujo del Motor**

  **A continuación, se presenta el pseudocódigo que describe el flujo del motor de simulación, mostrando la forma en que se seleccionan, ejecutan y finalizan los procesos dependiendo del modo de operación (automático o manual).**

  **Proceso FlujoMotor**

  ```
  Definir modo Como Cadena

  Definir proceso_actual Como Cadena

  Definir terminado Como Logico

  Escribir "Ingrese modo de simulacion (automatico/manual): "

  Leer modo

  Repetir

    // Seleccionar el siguiente proceso

    proceso_actual <- SeleccionarProceso()

    // Verificar si el proceso está terminado

    terminado <- VerificarTerminado(proceso_actual)

    Si terminado Entonces

      Escribir "Proceso ", proceso_actual, " está terminado. Se saca del ciclo."

    Sino

      Escribir "Ejecutando proceso: ", proceso_actual
  ```

```
            Si modo = "automatico" Entonces
                AvanzarAutomatico(proceso_actual)
            SiNo
                Escribir "Presione una tecla para avanzar..."
                Esperar Tecla
                AvanzarManual(proceso_actual)
            FinSi
        FinSi
    Hasta Que ProcesosTerminados() = Verdadero
    Escribir "Simulacion finalizada. Todos los procesos han terminado."
FinProceso
Funcion SeleccionarProceso() Como Cadena
    // Aquí va la lógica de selección del proceso
    SeleccionarProceso <- "P1"
FinFuncion
Funcion VerificarTerminado(proceso_actual Como Cadena) Como Logico
    // Devuelve VERDADERO si el proceso ya está en estado Terminated
    VerificarTerminado <- Falso
FinFuncion
SubProceso AvanzarAutomatico(proceso_actual Como Cadena)
    Escribir "El proceso ", proceso_actual, " avanza en modo automático..."
FinSubProceso
SubProceso AvanzarManual(proceso_actual Como Cadena)
    Escribir "El proceso ", proceso_actual, " avanza en modo manual..."
```

```
FinSubProceso

Funcion ProcesosTerminados() Como Logico

   // Retorna VERDADERO si ya no quedan procesos activos

   ProcesosTerminados <- Falso

FinFuncion
```

- Ejemplo de CSV exportado

| PID | Priority | PC | CpuRegisters | Syscalls | De | Para | Timestamp | Causa |
|---|---|---|---|---|---|---|---|---|
| 002 | 0 | 0 | {} | [] | New | Ready | 2025-09-08T02:33:37.004Z | auto |
| 002 | 0 | 1 | {AX":10}" | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:40.003Z"}]" | Ready | Running | 2025-09-08T02:33:40.003Z | auto |
| 002 | 0 | 1 | {AX":10}" | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:40.003Z"}]" | Running | Ready | 2025-09-08T02:33:43.004Z | auto |
| 002 | 0 | 2 | {AX":20}" | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:40.003Z"},{"type":"CPU_ASSIGN","at":" | Ready | Running | 2025-09-08T02:33:46.004Z | auto |
| 002 | 0 | 2 | {AX":20,"IO_WAIT":"2025-09-08T02:33:49.0 | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:40.003Z"},{"type":"CPU_ASSIGN","at":" | Running | Waiting | 2025-09-08T02:33:49.001Z | auto |
| 002 | 0 | 2 | {AX":20,"IO_WAIT":"2025-09-08T02:33:49.0 | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:40.003Z"},{"type":"CPU_ASSIGN","at":" | Waiting | Ready | 2025-09-08T02:33:52.006Z | auto |
| 002 | 0 | 3 | {AX":30,"IO_WAIT":"2025-09-08T02:33:49.0 | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:40.003Z"},{"type":"CPU_ASSIGN","at":" | Ready | Running | 2025-09-08T02:33:55.007Z | auto |
| 002 | 0 | 3 | {AX":30,"IO_WAIT":"2025-09-08T02:33:49.0 | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:40.003Z"},{"type":"CPU_ASSIGN","at":" | Running | Terminated | 2025-09-08T02:33:58.008Z | auto |
| 004 | 0 | 0 | {} | [] | New | Ready | 2025-09-08T02:33:03.992Z | manual |
| 004 | 0 | 1 | {AX":10}" | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:37.004Z"}]" | Ready | Running | 2025-09-08T02:33:37.004Z | auto |
| 004 | 0 | 1 | {AX":10}" | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:37.004Z"}]" | Running | Ready | 2025-09-08T02:33:40.003Z | auto |
| 004 | 0 | 2 | {AX":20}" | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:37.004Z"},{"type":"CPU_ASSIGN","at":" | Ready | Running | 2025-09-08T02:33:43.003Z | auto |
| 004 | 0 | 2 | {AX":20,"IO_WAIT":"2025-09-08T02:33:46.0 | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:37.004Z"},{"type":"CPU_ASSIGN","at":" | Running | Waiting | 2025-09-08T02:33:46.004Z | auto |
| 004 | 0 | 2 | {AX":20,"IO_WAIT":"2025-09-08T02:33:46.0 | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:37.004Z"},{"type":"CPU_ASSIGN","at":" | Waiting | Ready | 2025-09-08T02:33:49.001Z | auto |
| 004 | 0 | 3 | {AX":30,"IO_WAIT":"2025-09-08T02:33:46.0 | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:37.004Z"},{"type":"CPU_ASSIGN","at":" | Ready | Running | 2025-09-08T02:33:52.006Z | auto |
| 004 | 0 | 3 | {AX":30,"IO_WAIT":"2025-09-08T02:33:46.0 | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:37.004Z"},{"type":"CPU_ASSIGN","at":" | Running | Terminated | 2025-09-08T02:33:55.007Z | auto |
| 006 | 0 | 0 | {} | [] | New | Ready | 2025-09-08T02:33:40.003Z | auto |
| 006 | 0 | 1 | {AX":10}" | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:49.001Z"}]" | Ready | Running | 2025-09-08T02:33:49.001Z | auto |
| 006 | 0 | 1 | {AX":10,"END":"2025-09-08T02:33:52.006Z" | [{type":"CPU_ASSIGN","at":"2025-09-08T02:33:49.001Z"},{"type":"TERMINATE","at":"2 | Running | Terminated | 2025-09-08T02:33:52.006Z | auto |