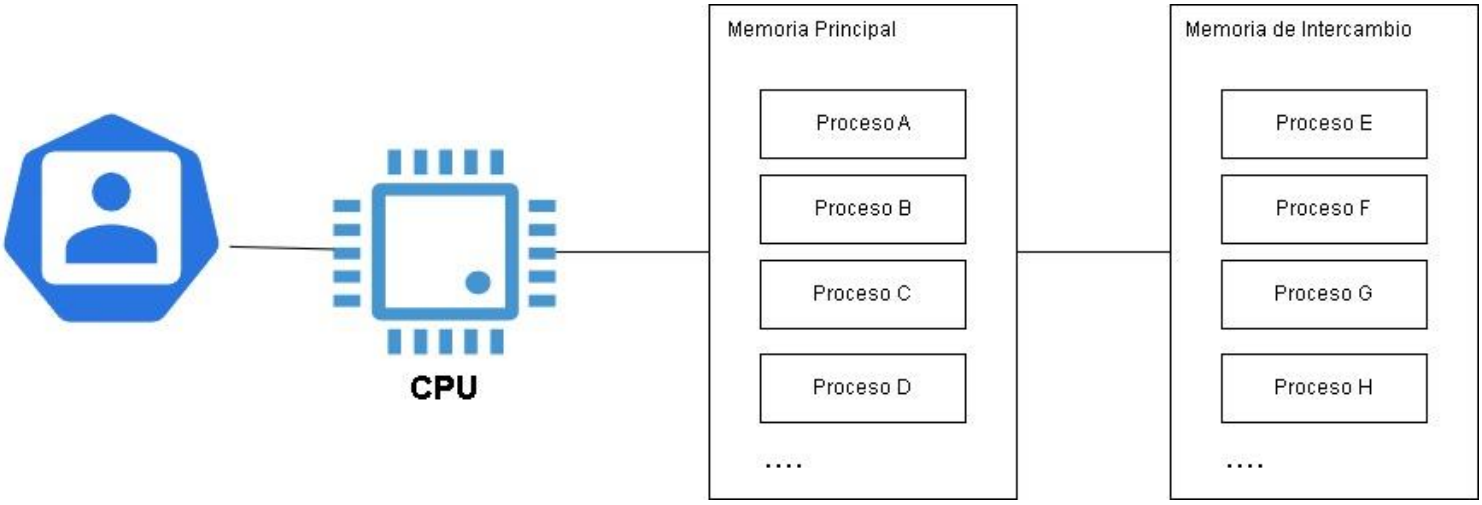
Memoria de intercambio (swapping)

Diego Fernández

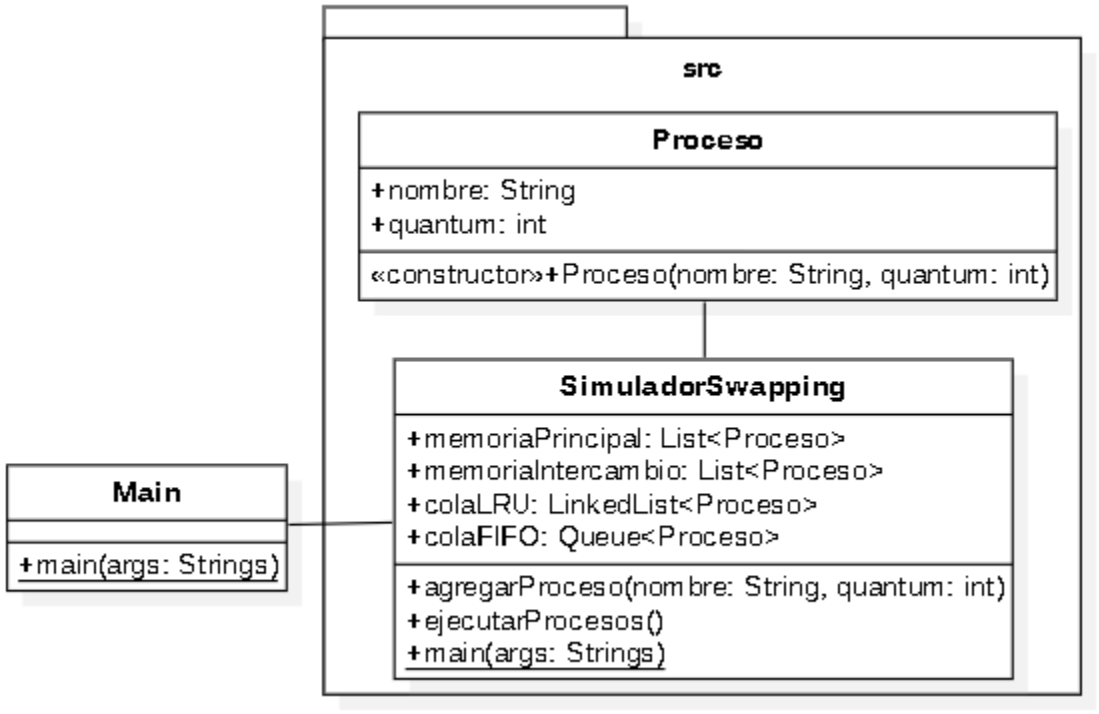
10/11/2023

V1.0

1. Definición de Arquitectura y Estructuras de Datos



Para la simulación de la memoria de intercambio (Swap) y la gestión de la reubicación de procesos, se propone una arquitectura basada en estructuras de datos específicas. Se utilizarán listas para representar tanto la memoria principal como la memoria de intercambio, facilitando la inserción y eliminación de procesos en ambas.



Memoria Principal:

- Se utiliza una lista para representar la memoria principal, donde cada elemento de la lista representa un bloque de memoria ocupado por un proceso.

$$List < Proceso > \ memoriaPrincipal \ = \ new \ ArrayList <> ();$$

- Las listas son apropiadas para representar la memoria principal debido a su facilidad para la inserción y eliminación de elementos.

Memoria de Intercambio:

- Similar a la memoria principal, se emplea otra lista para representar la memoria de intercambio.

$$List<Proceso> \ memoriaIntercambio = new \ ArrayList <> ();$$

- Las listas son apropiadas para representar la memoria de intercambio debido a su facilidad para la inserción y eliminación de elementos.

Proceso:

- Se ha definirá una clase “Proceso” para representar cada tarea. Esta clase incluye información relevante, como el nombre del proceso y el tiempo de cómputo asignado (quantum).

$$class \ Proceso \{$$
$$String \ nombre;$$
$$int \ quantum;$$
$$// \ Constructor, getters, setters, etc.$$
$$\}$$

Algoritmos de Selección de Procesos:

Para esto usaremos primeramente LRU ya que, es eficaz para mantener un historial del tiempo de acceso de los procesos. La LinkedList permite mantener este historial de manera eficiente, ya que los elementos pueden agregarse y eliminarse fácilmente en cualquier posición, además es adaptable a los patrones cambiantes de acceso a los procesos. Al utilizar una LinkedList, el algoritmo se ajusta automáticamente a los cambios en el orden de acceso, y finalmente también se usará FIFO ya que, es un algoritmo sencillo y fácil de entender. Utilizar una Queue para su implementación refleja claramente la política de "el primero en entrar es el primero en salir", además FIFO tiende a ser más estable en comparación con otros algoritmos. En situaciones donde se valora la predictibilidad y la equidad, FIFO es una elección sólida.

1. LRU (Least Recently Used):
   - Se implementará mediante una LinkedList que registrará el orden de acceso de los procesos.

     $$LinkedList < Proceso > \ colaLRU \ = \ new \ LinkedList <> ();$$
   - Cuando un proceso se ejecuta, se añade al final de la cola. La reubicación selecciona el proceso al frente de la cola, considerándolo el menos recientemente utilizado.
   - La LinkedList resulta idónea para LRU, ya que permite manejar fácilmente el orden de acceso mediante la adición y eliminación de elementos.

2. FIFO (First-In-First-Out):
   - Se utilizará una Queue simple para implementar FIFO.

     $$Queue < Proceso > \ colaFIFO \ = \ new \ LinkedList <> ();$$
   - Los procesos se añaden a la cola cuando se incorporan a la memoria principal. La reubicación selecciona el proceso al frente de la cola, siendo el que ha estado en la memoria por más tiempo.
   - El uso de una Queue se ajusta al principio de "el primero en entrar es el primero en salir", fundamental para FIFO.

Algoritmo:

Los algoritmos de selección de procesos se implementarán dentro de un bucle de ejecución de procesos en el programa principal. Para cada reubicación, se invocará el método correspondiente del algoritmo para seleccionar el proceso que debe trasladarse a la memoria de intercambio.

La elección de estructuras y algoritmos se ha guiado por la simplicidad y claridad. Esto facilita la comprensión del código y la mantenibilidad del sistema. Además, se ha priorizado la adaptabilidad a cambios en los patrones de acceso y en la cantidad de procesos. La estructura y los algoritmos elegidos permiten ajustarse eficientemente a las variaciones en el entorno de ejecución.

Se utilizará Java ya que:

- Java es conocido por su capacidad de portabilidad, lo que significa que se puede ejecutar código Java en diferentes plataformas sin cambios significativos. Esto es beneficioso para que el simulador sea compatible con varias plataformas.
- Java es un lenguaje de programación orientado a objetos, lo que facilita la modelización de conceptos del mundo real mediante clases y objetos. Esta característica es útil para representar entidades como procesos, memoria principal, memoria de intercambio, etc., como objetos.
- Java cuenta con una biblioteca estándar rica que proporciona una amplia gama de funciones y clases predefinidas. Esto puede acelerar el desarrollo, ya que no se tendrá que implementar funciones comunes desde cero.
- Java se ejecuta en una Máquina Virtual Java (JVM), lo que proporciona un entorno de ejecución seguro.
- Java tiene una amplia comunidad de desarrolladores y una gran cantidad de recursos de documentación en línea. Esto facilita la resolución de problemas y la obtención de ayuda en caso de dificultades.
- Java proporciona mecanismos robustos para el manejo de concurrencia, que pueden ser esenciales en un simulador donde los procesos pueden ejecutarse simultáneamente.