Implementación de un Simulador de Sistema Operativo Simplificado con Planificación de CPU, E/S y Administración de Memoria

Alvaro Raúl Quispe Condori, Christian Raul Mestas Zegarra, Luis Gustavo Sequeiros Condori, Yenaro Joel Noa Camino Universidad Nacional de San Agustín de Arequipa, Perú
aquispecondo@unsa.edu.pe, cmestasz@unsa.edu.pe, lsequeiros@unsa.edu.pe, ynoa@unsa.edu.pe

***Resumen*—resumen**

***Index Terms*—palabras**

I. Introducción

introduccion

II. Revisión de la literatura

En la siguiente sección se presenta una revisión de la literatura relacionada con los temas abordados en este informe, se tratan los temas de...

A. tema 1

tema 1 y su referencia [1]

III. Metodología

En esta sección se describe la metodología utilizada para el desarrollo del simulador de sistema operativo, incluyendo el diseño, implementación y los algoritmos empleados.

A. Diseño del simulador

A continuación, se detallan los aspectos relacionados con el diseño del simulador, específicamente, la arquitectura general del sistema, los diagramas de clases, el flujo de procesos y las técnicas de sincronización utilizadas.

A1. Arquitectura del sistema: el sistema fue diseñado en 2 sistemas principales, el simulador (c++) y el visualizador (python). el simulador se encarga de ejecutar la lógica del sistema operativo, mientras que el visualizador presenta los resultados de manera gráfica. el simulador se subdivide en los modulos de core (caracteristicas generales), cpu (planificación de CPU), io (planificación de E/S), memory (administración de memoria) y metrics (registro de eventos). el visualizador se subdivide en un modulo por grafico generado, los cuales son: diagrama de gantt, evolución de las colas, uso de memoria, tablas de páginas, asignaciones de los frames, operaciones de io, diagrama de gantt de io, cambios de contexto, resumenes estadísticos y distribucion de estados de procesos.

A2. Diagrama de clases: En la figura En la figura

A3. Flujo de procesos: de forma general, el simulador tiene el control absoluto sobre la ejecucion de los procesos, durante la simulacion, este avanza el reloj virtual y revisa los datos de entrada para determinar si algun nuevo proceso está ingresando en el tiempo actual, una vez un proceso ingresa, su estado es cambiado a listo, cuando el planificador de CPU selecciona un proceso para ejecutar, se intenta cargar sus paginas en memoria, efectivamente bloqueando el proceso por memoria, el administrador de memoria mantiene una cola interna y resuelve los fallos de pagina, tomando una unidad de tiempo por reemplazo, en este tiempo, como el proceso se bloqueó, el planificador de CPU puede haber tomado otro proceso, lo que significa que se necesita una forma de evitar el escenario en el que para el momento que el proceso bloqueado por memoria sea seleccionado de nuevo, ninguna de sus paginas estén disponibles, y entre a un bucle infinito, para esto, se realizo el uso del bit de referencia, el que marca las paginas cargadas por un proceso a punto de comenzar, haciendolas no elegibles para reemplazo hasta que su proceso padre entre y salga del estado running, ya sea por e/s, fin de rafaga o preemción, una vez el proceso salga de ejecución, sus paginas son marcadas como libres de nuevo, permitiendo su reemplazo, una vez un proceso termina todas sus ráfagas, se marca como terminado y ya no se le vuelve a considerar en la simulación. los bloqueos por io son mas sencillos, si un proceso se bloquea por io simplemente se mueve a la cola de bloqueados, y el planificador de io se encarga de realizar sus operaciones, cuando termine, el proceso regres a listo

A4. Técnicas de sincronización: Para cumplir con los requerimientos, cada proceso se ejecuta en un hilo, esto genera varias complicaciones al momento de mantener ”la verdad”sincronizada entre los diferentes modulos del simulador, para resolver esto, el simulador tiene como corazón una función ”tick”(execute_step) que es atomica mediante mutexes, esta funcion avanza el reloj virtual, y da permiso de ejecucion a los procesos que lo requieran, esperando a que estos marquen una variable de condicion una vez hayan terminado su tick. Los procesos, al ser lanzados, inician un hilo que simplemente espera a que el simulador le de permiso para ejecutar su tick, de esta forma se mantiene la sincronización entre los diferentes modulos del simulador, ya que todos los procesos solo pueden avanzar cuando el simulador lo permite. una vez el ”tick.es completado por el proceso, el simulador procede a darle un ”tick.a los modulos de memoria, y luego io,

usando una estrategia similar, finalmente haciendo registros de eventos y avanzando al siguiente ciclo. el manejo de variables compartidas se hace mediante variables atomicas, puesto que no se requiere de operaciones complejas sobre estas, y c++ ofrece un buen soporte para estas.

A5. Manejo de eventos y métricas: el simulador cuenta con un modulo de metricas que se encarga de registrar todos los eventos ocurridos durante la simulación, estos eventos son almacenados en memoria y luego exportados a un archivo en formato jsonl para su uso en el visualizador. un evento se considera como cualquier cambio de estado en un proceso, operación de io o fallo de pagina.

B. Implementación

B1. Clases globales: Las clases globales del simulador están declaradas en el directorio include/core, estas incluyen la clase Process, Burst y ConfigParser.
La clase Process representa un proceso en el sistema operativo simulado, y mantiene toda la información relevante sobre el proceso, además de su hilo de ejecución y métricas asociadas.
La clase Burst representa una ráfaga de CPU o E/S, y contiene información sobre la ráfaga como su tipo y duración.
La clase ConfigParser se encarga de leer y parsear el archivo de configuración de entrada.

B2. Planificador de CPU: El planificador de CPU está declarado en el directorio include/cpu, y contiene las clases CPUScheduler, Scheduler y sus implementaciones específicas como FCFS, SJF, RR y priority.
La clase CPUScheduler es la encargada de gestionar la planificación de CPU, y es basicamente la fuente de verdad para todo el simulador, ya que es la encargada de ejecutar la funcion ”tickz mantener todo el estado de procesos, dispositivos de e/s y memoria. la implementacion se realizó de esta forma porque se debian centralizar todos los datos en un solo lugar para evitar inconsistencias, y se eligio la planificación de CPU como el núcleo del simulador.
La clase Scheduler es una clase abstracta que define la interfaz para los diferentes algoritmos de planificación de CPU.
Las implementaciones específicas de los algoritmos de planificación heredan de la clase Scheduler y implementan el método de selección de procesos según el algoritmo correspondiente.

B3. Planificador de E/S: El planificador de E/S está declarado en el directorio include/io, y contiene las clases IORequest, IODevice, IOManager, IOScheduler y sus implementaciones específicas.
La clase IORequest representa una solicitud de E/S realizada por un proceso, esta es la que inicia todo el flujo de E/S en el simulador.
La clase IODevice representa un dispositivo de E/S en el sistema operativo simulado y es el encargado de cumplir con las solicitudes de E/S y registro de metricas.
La clase IOManager contiene la lógica para gestionar múltiples dispositivos de E/S, sin embargo, en la implementación actual, se decidió utilizar un solo dispositivo por razones de tiempo, la implementación es actualmente usada por el simulador, pero no es configurable y no está probada, asi que usar multiples dispositivos de e/s podria generar resultados inesperados.
La clase IOScheduler es una clase abstracta que define la interfaz para los diferentes algoritmos de planificación de E/S.
Las implementaciones específicas de los algoritmos de planificación heredan de la clase IOScheduler y implementan el método de selección de solicitudes según el algoritmo correspondiente.

B4. Administrador de memoria: El administrador de memoria está declarado en el directorio include/memory, y contiene las clases MemoryManager, Page, ReplacementAlgorithm y sus implementaciones específicas como FIFO, LRU y NRU.
La clase MemoryManager es la encargada de gestionar la memoria del sistema operativo simulado, incluyendo la asignación y liberación de marcos de página, así como la recepcion y solucion de fallos de página.
La clase Page representa una página de memoria en el sistema operativo simulado, y sirve como contenedor de la información relevante sobre la página.
La clase ReplacementAlgorithm es una clase abstracta que define la interfaz para los diferentes algoritmos de reemplazo de páginas.
Las implementaciones específicas de los algoritmos de reemplazo heredan de la clase ReplacementAlgorithm y implementan el método de selección de páginas según el algoritmo correspondiente.

B5. Registro de eventos: El colector de eventos está declarado en el directorio include/metrics, y contiene la clase MetricsCollector.
La clase MetricsCollector es la encargada de registrar y almacenar los eventos ocurridos durante la simulación y exportarlo en diferentes formatos para su posterior análisis y visualización.
una instancia de metricscollector es instanciada e inyectada en los diferentes modulos del simulador para que estos puedan registrar todos los eventos del simulador.
en la implementacion final, solo se hace uso del formato jsonl, puesto que es el formato mas sencillo de manejar para el visualizador.

B6. Visualizador de resultados: El visualizador de resultados está implementado en Python y se encuentra en el directorio visualization, esta hace uso de la libreria seaborn para la generación de gráficos.
el visualizador contiene las clases DataLoader, Visualizer, BaseGenerator y sus implementaciones específicas para cada tipo de gráfico.
La clase DataLoader es la encargada de cargar y parsear los datos del archivo de eventos generado por el simulador.
La clase Visualizer es la encargada de gestionar la generación de gráficos a partir de los datos cargados.
La clase BaseGenerator es una clase abstracta que define la interfaz para los diferentes tipos de gráficos.
Las implementaciones específicas de los gráficos heredan

de la clase BaseGenerator y implementan el método de generación de gráficos según el tipo correspondiente.

B7. Documentación, CLI, tests y repositorio: El proyecto cuenta con documentación generada mediante Doxygen, la cual se genera en el directorio docs siguiendo las instrucciones de README.md.
El proyecto tambien cuenta con una interfaz de línea de comandos (CLI) ejecutables como los archivos build/bin/os_simulator y python -m visualization, los cuales permiten ejecutar el simulador y el visualizador respectivamente, siguiendo las instrucciones de README.md.
El proyecto cuenta con tests unitarios implementados utilizando la biblioteca Catch2, los cuales se encuentran en el directorio tests, y pueden ser ejecutados mediante el comando build/bin/tests, siguiendo las instrucciones de README.md.
El código fuente del proyecto se encuentra alojado en un repositorio de GitHub en la siguiente URL:
`https://github.com/gustadev24/os-simulator`

C. Algoritmos

A continuacion se describen los algoritmos implementados en el simulador, tanto para la planificación de CPU y E/S, como para el reemplazo de páginas en la administración de memoria.

C1. Algoritmos de planificación de CPU: Los algoritmos de planificación de CPU implementados en el simulador son:
FCFS: este algoritmo hace uso de una cola FIFO para almacenar los procesos listos, y cada vez que tiene que elegir, selecciona el proceso que ingreso primero para su ejecución.
SJF: este algoritmo ordena los procesos listos según la duración de su próxima ráfaga de CPU, y cada vez que tiene que elegir, selecciona el proceso con la ráfaga más corta para su ejecución.
RR: este algoritmo hace uso de una cola FIFO para almacenar los procesos listos, y cada vez que tiene que elegir, selecciona el proceso que ingreso primero para su ejecución, pero solo le permite ejecutar por un tiempo determinado (quantum), luego de esto, el proceso es preemptado y se vuelve a colocar al final de la cola.
Priority: este algoritmo ordena los procesos listos según su prioridad, y cada vez que tiene que elegir, selecciona el proceso con la mayor prioridad para su ejecución.

Para la planificacion de E/S, se implementaron los algoritmos FCFS y RR, siguiendo una logica similar a la de la planificacion de CPU.

C2. Algoritmos de reemplazo de páginas: Los algoritmos de reemplazo de páginas implementados en el simulador son:
FIFO: este algoritmo mantiene una cola FIFO de las páginas cargadas en memoria, y cada vez que se necesita reemplazar una página, selecciona la página que ingreso primero para su reemplazo.
LRU: este algoritmo mantiene un registro del último acceso de cada página cargada en memoria, y cada vez que se necesita reemplazar una página, selecciona la página que no ha sido accedida por el mayor tiempo para su reemplazo.
NRU: este algoritmo clasifica las páginas cargadas en memoria en cuatro categorías según sus bits de referencia y modificación, y cada vez que se necesita reemplazar una página, selecciona una página al azar de la categoría más baja para su reemplazo. en la implementacion, tecnicamente solo existen dos categorias, puesto que los requerimientos no consideran modificaciones a la memoria como parte de la simulacion.
Optimo: este algoritmo selecciona la página que no será utilizada por el mayor tiempo en el futuro, ya que no se tienen accesos individuales a las paginas, .[el] futuro"se considera como 1. reemplazar paginas de procesos terminados, 2. reemplazar paginas de procesos bloqueados por e/s, ordenados de mayor a menor tiempo restante de e/s, 3. reemplazar cualquier pagina que no este bloqueada

IV. Resultados

A. Casos de prueba

B. Análisis de resultados

V. Discusión y conclusiones

Referencias

[1] F. Mittelbach and M. Goossens, *The LaTeX Companion*.