

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

1^{er} Alvaro Raúl Quispe Condori

Universidad Nacional de San Agustín de Arequipa
Arequipa, Perú
aquispecondo@unsa.edu.pe

3^{er} Luis Gustavo Sequeiros Condori

Universidad Nacional de San Agustín de Arequipa
Arequipa, Perú
lsequeiros@unsa.edu.pe

2^{do} Christian Raul Mestas Zegarra

Universidad Nacional de San Agustín de Arequipa
Arequipa, Perú
cmestasz@unsa.edu.pe

4^{to} Yenaro Joel Noa Camino

Universidad Nacional de San Agustín de Arequipa
Arequipa, Perú
ynoa@unsa.edu.pe

Resumen—El presente trabajo elabora un simulador hecho con C++ 17 de un sistema operativo que integra la unidad central de procesamiento, la gestión de entrada y salida (E/S) y la administración de memoria virtual por paginación. Así mismo, un módulo de Python realiza gráficos a partir de los registros generados por el simulador para el análisis del comportamiento del sistema. Se realizó un diseño modular para la extensión de componentes y que permita alternar entre diferentes combinaciones de algoritmos.

Index Terms—planificación de procesos, memoria virtual, paginación, simulación de sistemas operativos, algoritmos de reemplazo

I. INTRODUCCIÓN

Los sistemas operativos [1] son responsables de la gestión de los recursos del hardware y de proveer servicios a las aplicaciones que se ejecutan sobre él. Entre las principales funciones de un sistema operativo se pueden mencionar la planificación del uso del CPU, la gestión de memoria para la asignación de la misma a los programas y la coordinación de las operaciones de entrada y salida. La coordinación de estos tres factores determina el rendimiento del sistema en general.

El presente trabajo desarrolla un simulador del comportamiento de los subsistemas mencionados anteriormente. El componente del CPU implementa el algoritmo First Come First Served (FCFS), Shortest Job First (SJF), Round Robin y la planificación por prioridad. El gestor de memoria utiliza paginación con algoritmos de reemplazo entre First In First Out (FIFO), Least Recently Used (LRU), Not Recently Used (NRU) y el algoritmo óptimo. Por último, el planificador de E/S soporta FCFS y Round Robin. Cada proceso se ejecuta en un hilo independiente para replicar la concurrencia de un sistema operativo.

El documento se organiza de la siguiente manera: la Sección II presenta los conceptos teóricos fundamentales; la Sección III describe las herramientas utilizadas; la Sección IV detalla

la metodología, incluyendo arquitectura, implementación y algoritmos; la Sección V muestra los resultados experimentales; y la Sección VI expone las conclusiones.

II. REVISIÓN DE LA LITERATURA

Esta sección presenta los conceptos teóricos necesarios para comprender el funcionamiento del simulador desarrollado.

II-A. Sistemas operativos y procesos

Un sistema operativo es el intermediario entre usuario y hardware que permite la ejecución de otros programas [1]. Se encarga de la planificación de recursos, la gestión de una interfaz de usuario y la ejecución de los procesos de aplicaciones.

Por su parte, un proceso es un programa que incluye el código del programa, su actividad en el contador de programa, registros en el procesador y una sección con datos de variables globales [1].

II-B. Planificación de CPU y algoritmos de planificación

Es el mecanismo mediante el cual el sistema operativo decide qué proceso utilizará el procesador en algún momento dado [2]. Los algoritmos de planificación se clasifican en apropiativos, que permiten interrumpir un proceso en ejecución, y no apropiativos, que permiten que un proceso se ejecute hasta que termine o se bloquee voluntariamente.

El algoritmo FCFS [1] atiende los procesos en el orden en que llegan a la cola de listos. SJF [2] selecciona el proceso con la ráfaga de CPU más corta. Round Robin [3] asigna un intervalo de tiempo fijo llamado quantum [3] a cada proceso de forma cíclica. La planificación por prioridad [2] asigna el procesador al proceso con mayor prioridad.

II-C. Memoria virtual y paginación

La memoria virtual [1] permite ejecutar procesos que superan la memoria física. El sistema mantiene en memoria solo las partes del proceso que se están usando, mientras el resto permanece en almacenamiento secundario.

Por su parte, la paginación [3] divide el espacio de direcciones en páginas de tamaño fijo que se mapean a marcos en memoria física. Cuando un proceso accede a una página que no está cargada, ocurre un fallo de página [1] y el sistema debe traerla desde el almacenamiento secundario.

Los algoritmos de reemplazo deciden qué página retirar cuando la memoria está llena. FIFO [3] reemplaza la página más antigua. LRU [1] retira la página que lleva más tiempo sin usarse. NRU [3] clasifica las páginas según sus bits de referencia y modificación. El algoritmo óptimo [1] elige la página que no se usará durante más tiempo en el futuro.

II-D. Gestión de entrada/salida

La gestión de E/S [2] coordina el acceso a los dispositivos periféricos. Las operaciones de E/S son mucho más lentas que las del CPU, así que los procesos se bloquean mientras esperan su finalización. Los algoritmos de planificación de E/S determinan el orden en que se atienden las solicitudes pendientes.

II-E. Concurrencia y sincronización

La concurrencia [2] permite que varios procesos o hilos avancen al mismo tiempo. Cuando comparten recursos, se necesitan mecanismos de sincronización para mantener la consistencia de los datos.

Un mutex [3] garantiza exclusión mutua, haciendo que solo un hilo entre a una sección crítica. Las variables de condición [1] permiten que un hilo espere hasta que se cumpla una condición específica. La inanición [2] ocurre cuando un proceso nunca obtiene los recursos necesarios para continuar.

III. HERRAMIENTAS UTILIZADAS

Además de C++ [4] y Python [5], que fueron los lenguajes de programación principales del proyecto, se utilizaron las siguientes herramientas para el desarrollo, compilación, pruebas y documentación del sistema.

CMake [6] es un sistema de construcción multiplataforma que genera archivos de configuración para diferentes entornos de compilación. En este proyecto se utilizó para gestionar las dependencias y configurar el proceso de compilación del simulador.

Ninja [7] es un sistema de compilación diseñado para ejecutar compilaciones de manera eficiente. Se empleó como backend de CMake para acelerar el proceso de compilación incremental durante el desarrollo.

Catch2 [8] es un framework de pruebas unitarias para C++. Se utilizó para implementar y ejecutar las pruebas que verifican el correcto funcionamiento de los algoritmos de planificación y reemplazo de páginas.

Just [9] es un ejecutor de comandos que permite definir tareas frecuentes en un archivo de configuración. Se empleó

para automatizar operaciones comunes como compilación, ejecución de pruebas y generación de documentación.

Seaborn [10] es una biblioteca de visualización estadística para Python basada en Matplotlib. Se utilizó en el módulo de visualización para generar los diagramas de Gantt [11], gráficos de evolución de colas y otras representaciones del comportamiento del sistema.

Doxxygen [12] es un generador de documentación que extrae información de los comentarios del código fuente. Se empleó para generar la documentación técnica del proyecto en formatos HTML y PDF.

LaTeX [13] es un sistema de composición tipográfica orientado a la creación de documentos técnicos y científicos. Se utilizó para la redacción de este informe siguiendo el formato IEEE.

PlantUML [14] es una herramienta que permite crear diagramas UML a partir de descripciones textuales. En conjunto con clang-uml [15], que genera automáticamente diagramas de clases desde código C++, se utilizaron para documentar la arquitectura del simulador y del visualizador mediante diagramas de clases.

IV. METODOLOGÍA

Esta sección describe el diseño del simulador, los detalles de implementación de cada módulo y los algoritmos empleados para planificación y reemplazo de páginas.

IV-A. Arquitectura del sistema

El sistema se compone de dos módulos principales: un simulador desarrollado en C++17 [4] y un visualizador implementado en Python [5]. El primero ejecuta la lógica del sistema operativo, mientras que el segundo procesa los datos generados para producir gráficos que facilitan el análisis del comportamiento.

El simulador se subdivide en cinco componentes. El módulo Core contiene las clases fundamentales que representan procesos y ráfagas de ejecución. El módulo CPU implementa el planificador central y los algoritmos de selección de procesos. El módulo IO gestiona los dispositivos de E/S y sus planificadores. El módulo Memory administra los marcos físicos y los algoritmos de reemplazo de páginas. El módulo Metrics recopila eventos durante la simulación para su posterior análisis. La figura 1 muestra la estructura de clases del simulador.

El visualizador procesa archivos en formato JavaScript Object Notation Lines (JSONL) [16] generados por el simulador. Cada generador especializado produce un tipo de gráfico: diagramas de Gantt [11] para CPU y E/S, evolución temporal de las colas de procesos, uso de memoria, estado de las tablas de páginas, asignación de marcos físicos, cambios de contexto y distribución de estados. La figura 2 muestra la estructura de clases del visualizador.

IV-B. Modelo de procesos

Un proceso [1] en el simulador se define mediante su identificador, tiempo de llegada, secuencia de ráfagas, prioridad y número de páginas requeridas. La secuencia de ráfagas alterna



Figura 1. Diagrama UML de las clases del simulador.

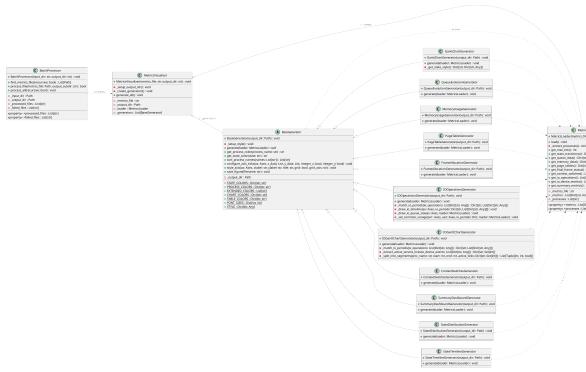


Figura 2. Diagrama UML de las clases del visualizador.

entre períodos de uso de CPU y operaciones de E/S. Cada proceso atraviesa seis estados durante su ciclo de vida: nuevo, listo, en espera de memoria, en ejecución, bloqueado por E/S y terminado.

Cuando un proceso llega al sistema, se encuentra en estado nuevo. El planificador lo mueve a la cola de listos cuando corresponde según su tiempo de llegada. Al ser seleccionado para ejecución, el administrador de memoria verifica que sus páginas estén cargadas. Si alguna página no está presente, el proceso pasa a estado de espera de memoria hasta que el fallo se resuelva. Una vez cargadas las páginas, el proceso entra en ejecución y consume su ráfaga de CPU. Si la ráfaga siguiente

es de E/S, el proceso se bloquea hasta que la operación complete. Al finalizar todas sus ráfagas, el proceso termina.

IV-C. Flujo de ejecución

El simulador avanza mediante un reloj virtual donde cada unidad de tiempo se define como tick, donde se ejecuta una secuencia de operaciones coordinadas. Primero se verifican las llegadas de nuevos procesos según los tiempos definidos en el archivo de entrada. Luego el planificador de CPU selecciona el siguiente proceso a ejecutar de acuerdo con el algoritmo configurado. Si el proceso seleccionado requiere páginas que no están en memoria, se encola una solicitud de carga y el proceso se bloquea.

El administrador de memoria resuelve un fallo de página por tick. Cuando todas las páginas de un proceso están cargadas, este puede continuar su ejecución. Durante la ejecución, el proceso consume una unidad de tiempo de su ráfaga actual. Si completa una ráfaga de CPU y la siguiente es de E/S, se envía una solicitud al planificador de E/S y el proceso se bloquea. El planificador de E/S atiende las solicitudes según su algoritmo configurado. Al completar una operación de E/S, el proceso regresa a la cola de listos. La Figura 3 ilustra este proceso.

IV-D. Sincronización concurrente

Cada proceso se ejecuta en un hilo independiente para replicar el comportamiento concurrente de un sistema operativo. Esta decisión de diseño introduce la necesidad de mecanismos de sincronización para mantener la consistencia del estado global.

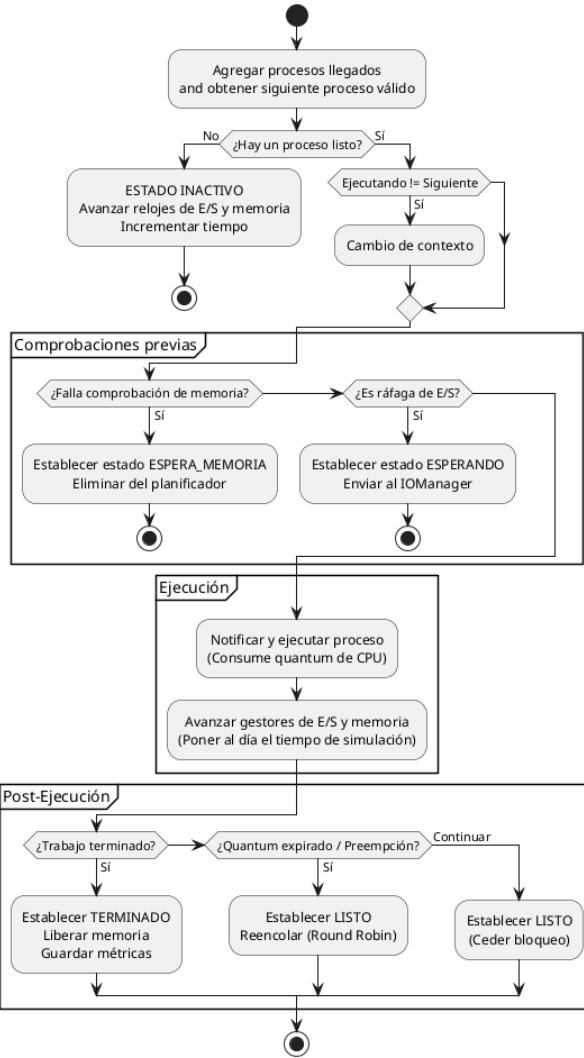


Figura 3. Flujo de ejecución de un tick en el simulador, mostrando la interacción entre los módulos de CPU, memoria y E/S.

La función de tick está protegida por un mutex [3] que coordina la ejecución. Esta cumple con sus tareas previas, notifica al proceso activo que puede ejecutar su porción de trabajo y espera mediante una variable de condición [1] hasta que el proceso señale que completó su tick. Posteriormente, la función realiza su trabajo posterior a la ejecución del paso del proceso.

Las variables compartidas entre hilos utilizan tipos atómicos de C++ cuando las operaciones son simples lecturas o escrituras. Para operaciones más complejas que requieren consistencia entre múltiples variables, se emplean secciones protegidas por mutex.

IV-E. Prevención de inanición por memoria

Un problema que puede surgir durante la simulación es la inanición [2] de un proceso que espera cargar sus páginas en memoria. Cuando un proceso es seleccionado para ejecución, el administrador de memoria intenta cargar sus páginas. Sin

embargo, mientras el proceso espera, el planificador puede seleccionar otro proceso, cuyas páginas podrían reemplazar las que el primer proceso acaba de cargar.

Para evitar este escenario, el simulador implementa un mecanismo basado en el bit de referencia. Cuando las páginas de un proceso se cargan en preparación para su ejecución, se marcan como protegidas. Estas páginas no pueden ser seleccionadas como víctimas hasta que el proceso termine su ráfaga actual, ya sea por completar su tiempo de CPU, por un bloqueo de E/S o por preempción en algoritmos apropiativos. Una vez que el proceso sale del estado de ejecución, sus páginas se desprotegen y quedan disponibles para reemplazo.

IV-F. Algoritmos de planificación de CPU

El simulador implementa cuatro algoritmos de planificación de CPU [2]. FCFS [1] atiende los procesos en el orden en que llegan a la cola de listos. Este algoritmo no es apropiativo, por lo que un proceso en ejecución continúa hasta completar su ráfaga. SJF [2] selecciona el proceso cuya próxima ráfaga de CPU sea la más corta. En la implementación actual, SJF tampoco es apropiativo.

Round Robin [3] asigna a cada proceso un intervalo de tiempo fijo denominado quantum [3]. Cuando un proceso agota su quantum sin completar su ráfaga, es preemptado y reinsertado al final de la cola de listos. El planificador por prioridad [2] selecciona el proceso con mayor prioridad, representada por el menor valor numérico. Este algoritmo es apropiativo: si llega un proceso con mayor prioridad que el actual, se produce una preempción. El Cuadro I resume las características de cada algoritmo.

Cuadro I
ALGORITMOS DE PLANIFICACIÓN DE CPU IMPLEMENTADOS.

Algoritmo	Criterio de selección	Apropiativo
FCFS	Orden de llegada	No
SJF	Menor ráfaga	No
Round Robin	Turno con quantum	Sí
Por prioridad	Mayor prioridad	Sí

IV-G. Algoritmos de reemplazo de páginas

El administrador de memoria implementa cuatro algoritmos de reemplazo. FIFO [3] mantiene un registro del orden de llegada de las páginas a memoria y selecciona la más antigua para reemplazo. LRU [1] registra el último acceso a cada página y selecciona aquella que no ha sido utilizada durante más tiempo.

NRU [3] clasifica las páginas según sus bits de referencia y modificación. En la implementación actual, dado que la simulación no contempla modificaciones a memoria, efectivamente solo existen dos categorías: páginas referenciadas y no referenciadas. El algoritmo selecciona una página de la categoría más baja disponible.

El algoritmo óptimo [1] tradicionalmente selecciona la página que no será accedida durante más tiempo en el futuro. En sistemas reales, esta información no está disponible. En el

simulador desarrollado, aunque se conoce la secuencia futura de ráfagas, no se simulan accesos individuales a páginas de memoria, ya que los requerimientos establecen que un proceso necesita todas sus páginas cargadas para ejecutarse. Por lo tanto, el algoritmo óptimo no puede evaluar accesos futuros a páginas específicas. La implementación utiliza una heurística que aproxima este comportamiento: primero considera como víctimas las páginas de procesos terminados, luego las de procesos bloqueados por E/S ordenados por tiempo restante de mayor a menor, y finalmente cualquier página que no esté protegida. El Cuadro II resume los algoritmos implementados.

Cuadro II
ALGORITMOS DE REEMPLAZO DE PÁGINAS IMPLEMENTADOS.

Algoritmo	Criterio de selección
FIFO	Página con mayor tiempo en memoria
LRU	Página no accedida por más tiempo
NRU	Página con menor categoría según bit de referencia
Óptimo	Heurística basada en estado de procesos

IV-H. Planificación de E/S

El módulo de E/S [2] gestiona las solicitudes de acceso a dispositivos periféricos. La implementación actual utiliza un único dispositivo denominado disco. Cuando un proceso completa una ráfaga de CPU y su siguiente ráfaga es de E/S, se genera una solicitud que se encola en el planificador del dispositivo.

Los algoritmos disponibles son FCFS y Round Robin, con funcionamiento análogo a sus contrapartes de CPU. En FCFS, las solicitudes se atienden en orden de llegada y cada operación se completa antes de comenzar la siguiente. En Round Robin, cada solicitud recibe un quantum de tiempo configurable; si no completa en ese intervalo, se reencola y se atiende la siguiente solicitud pendiente.

La arquitectura del módulo de E/S está diseñada para soportar múltiples dispositivos, aunque la configuración actual emplea únicamente uno. Cada dispositivo mantiene su propia cola de solicitudes pendientes y puede configurarse con un planificador independiente. Esta implementación se canceló debido a falta de tiempo, y, aunque el administrador existe en el código, su funcionalidad multi-dispositivo no está probada y no tiene garantía de funcionar.

IV-I. Recolección de métricas

El módulo de métricas registra eventos durante toda la simulación. Los eventos se agrupan por tick e incluyen información sobre el estado de la CPU, operaciones de memoria, actividad de E/S y transiciones de estado de los procesos. Al finalizar la simulación, los datos se exportan en formato JSONL, donde cada línea representa un objeto JSON independiente correspondiente a un tick.

El visualizador en Python lee estos archivos y genera gráficos mediante la biblioteca Seaborn [10]. Los diagramas de Gantt, como los mostrados en las Figuras 4 y 5, presentan

la asignación temporal de CPU y dispositivos de E/S a los procesos. Los gráficos de evolución de colas, como el ilustrado en la Figura 6, muestran cómo cambia la cantidad de procesos en cada estado a lo largo del tiempo.

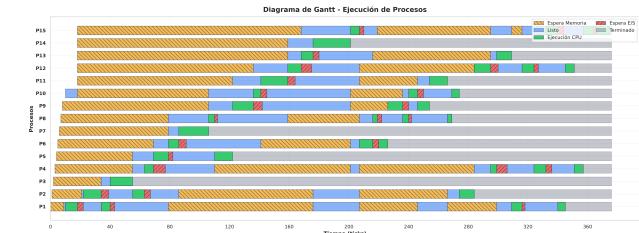


Figura 4. Ejemplo de diagrama de Gantt generado por el visualizador, mostrando la asignación de CPU.

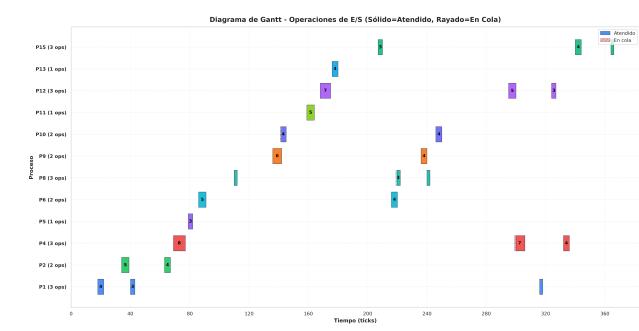


Figura 5. Ejemplo de diagrama de Gantt generado por el visualizador, mostrando la asignación de IO.

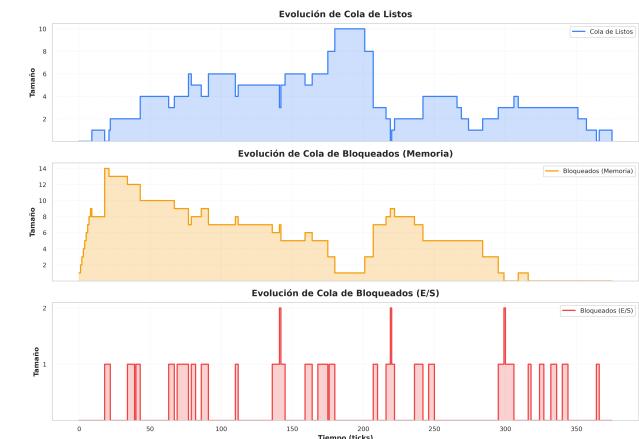


Figura 6. Ejemplo de gráfico de evolución de colas, mostrando la cantidad de procesos en cada estado a lo largo del tiempo.

IV-J. Configuración del sistema

El simulador lee dos archivos de configuración al inicio. El archivo de configuración general especifica parámetros globales como el número de marcos de memoria disponibles, el tamaño de cada marco, los algoritmos a utilizar y los valores de quantum. El archivo de procesos define cada proceso

con su identificador, tiempo de llegada, secuencia de ráfagas, prioridad y número de páginas requeridas.

El Listado 1 muestra un ejemplo de archivo de configuración. En este caso, el sistema dispone de 64 marcos de memoria de 4096 bytes cada uno, utiliza Round Robin para planificación de CPU con quantum de 4 unidades, LRU para reemplazo de páginas y FCFS para planificación de E/S.

```
total_memory_frames=64
frame_size=4096
scheduling_algorithm=RoundRobin
page_replacement_algorithm=LRU
io_scheduling_algorithm=FCFS
quantum=4
io_quantum=4
```

Listing 1. Ejemplo de archivo de configuración del simulador.

El Listado 2 ilustra la definición de tres procesos. El proceso P1 llega en el tiempo 0, tiene prioridad 1, requiere 4 páginas y ejecuta una ráfaga de CPU de 4 unidades, seguida de una operación de E/S de 3 unidades y una ráfaga final de CPU de 5 unidades.

```
# PID llegada rafagas prioridad paginas
P1 0 CPU(4),E/S(3),CPU(5) 1 4
P2 2 CPU(6),E/S(2),CPU(3) 2 5
P3 4 CPU(8) 3 6
```

Listing 2. Ejemplo de archivo de definición de procesos.

IV-K. Interfaz de línea de comandos

El simulador se ejecuta desde la terminal y acepta parámetros para especificar los archivos de entrada y salida. La opción -f indica el archivo de procesos, -c el archivo de configuración y -m el archivo donde se guardarán las métricas. El visualizador se invoca como un módulo de Python y soporta procesamiento individual de archivos o por lotes de un directorio completo.

El proyecto incluye pruebas unitarias implementadas con la biblioteca Catch2 [8] que verifican el correcto funcionamiento de los algoritmos de planificación y reemplazo. La documentación del código se genera mediante Doxygen [12] y está disponible en formato HTML y PDF. El código fuente del proyecto se encuentra disponible en el repositorio <https://github.com/gustadev24/os-simulator>, donde el archivo README contiene instrucciones detalladas sobre instalación, compilación, ejecución y generación de documentación.

V. RESULTADOS

Para evaluar el comportamiento del simulador se diseñaron cuatro casos de prueba que representan diferentes escenarios de carga y demuestran tanto el funcionamiento correcto del sistema como situaciones problemáticas conocidas en sistemas operativos. Cada caso utiliza una configuración específica de procesos y parámetros del sistema. Los resultados se analizaron mediante los diagramas de Gantt y gráficos de uso de memoria generados por el visualizador. Los archivos de configuración y procesos utilizados se incluyen en el Anexo A.

V-A. Caso 1: Carga base

El primer caso representa una carga de trabajo de referencia con 10 procesos que llegan de forma escalonada. Cada proceso ejecuta ráfagas de CPU de duración moderada intercaladas con operaciones de E/S. La configuración utiliza Round Robin con quantum de 4 unidades, LRU para reemplazo de páginas y 32 marcos de memoria disponibles.

La Figura 7 muestra la ejecución de este escenario. Se observa una distribución equitativa del tiempo de CPU entre los procesos gracias al algoritmo Round Robin. Los períodos de bloqueo por E/S son visibles como interrupciones en la ejecución de cada proceso. El sistema maneja adecuadamente la presión de memoria, utilizando LRU para seleccionar las páginas víctima de manera eficiente. Este caso sirve como línea base para comparar con los escenarios problemáticos siguientes.

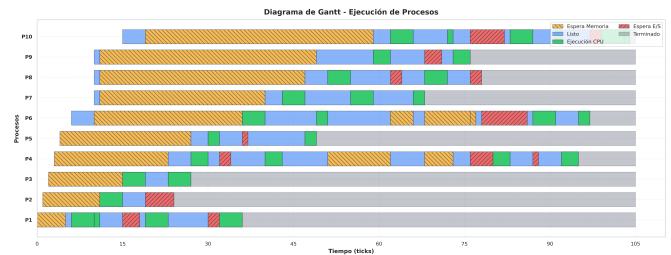


Figura 7. Diagrama de Gantt del caso base mostrando la ejecución de 10 procesos con carga moderada.

V-B. Caso 2: Round Robin ineficiente con ráfagas largas

Este caso demuestra una configuración inadecuada del algoritmo Round Robin. Se ejecutan 10 procesos con ráfagas de CPU extremadamente largas, entre 12 y 30 unidades de tiempo. El quantum se configura deliberadamente bajo, con solo 2 unidades de tiempo, mientras que las ráfagas requieren entre 6 y 15 quantums para completarse.

La Figura 8 ilustra el problema de esta configuración. El diagrama muestra una alta frecuencia de cambios de contexto: cada proceso es interrumpido constantemente antes de completar su trabajo. Por ejemplo, el proceso P5 con una ráfaga de 30 unidades requiere 15 preempciones para completarse. Esta fragmentación incrementa significativamente la sobrecarga del sistema y prolonga el tiempo de finalización de todos los procesos. El caso demuestra que la elección del quantum debe considerar la naturaleza de las ráfagas: un quantum muy pequeño para procesos con ráfagas largas resulta en una pérdida considerable de eficiencia.

V-C. Caso 3: Inanición severa con FCFS

El tercer caso expone el problema del efecto convoy en el algoritmo FCFS. Un proceso con una ráfaga inicial de 60 unidades de CPU (P1) llega primero al sistema. Inmediatamente después llegan seis procesos rápidos (P2-P6) con ráfagas cortas de entre 1 y 4 unidades. Finalmente, otro proceso pesado (P7) con 30 unidades de ráfaga inicial se suma a la cola.

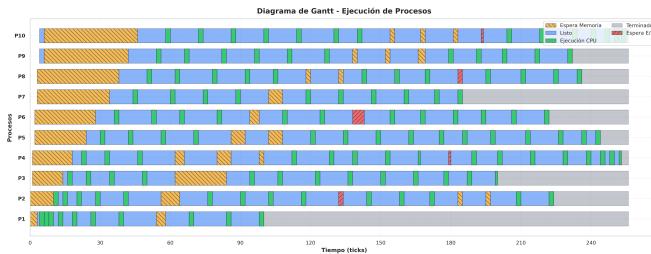


Figura 8. Diagrama de Gantt del caso Round Robin ineficiente, mostrando la fragmentación excesiva causada por un quantum inadecuado.

La Figura 9 presenta la ejecución de este escenario. Los procesos P2 a P6, que podrían completarse en pocos ticks, deben esperar 60 unidades mientras P1 monopoliza el procesador. A pesar de tener ráfagas combinadas de solo 14 unidades, estos procesos experimentan tiempos de espera desproporcionados. El proceso P7 agrava la situación al añadir otra ráfaga larga después de los procesos cortos. Este caso ilustra por qué FCFS no es apropiado para sistemas interactivos o cargas mixtas: los procesos cortos sufren inanición temporal severa cuando quedan atrapados detrás de procesos largos.

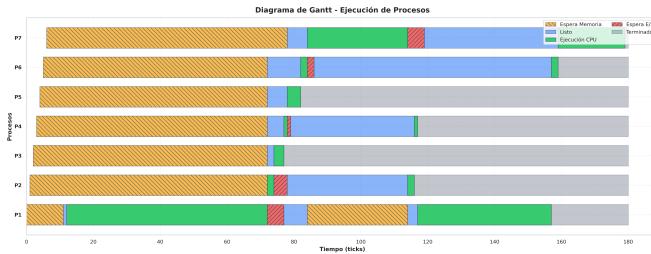


Figura 9. Diagrama de Gantt del caso FCFS mostrando inanición severa de procesos cortos.

V-D. Caso 4: Estrés de memoria y thrashing

El cuarto caso evalúa el comportamiento del sistema bajo presión extrema de memoria. Se ejecutan 7 procesos que alternan frecuentemente entre ráfagas de CPU y operaciones de E/S, con requerimientos de memoria que suman 42 páginas. La configuración dispone de solo 18 marcos de memoria, lo que genera constantes reemplazos de páginas. Cada transición entre estados provoca fallos de página cuando el proceso retoma su ejecución.

La Figura 10 muestra los resultados de este escenario. El diagrama de Gant presentó períodos frecuentes de espera por memoria entre las ejecuciones, ya que los procesos deben recargar sus páginas constantemente. El patrón de múltiples transiciones CPU-E/S amplifica el problema: cada vez que un proceso vuelve de una operación de E/S, es probable que sus páginas hayan sido reemplazadas por otros procesos.

Es importante notar que reducir la memoria a 16 marcos en este escenario provoca un colapso del sistema por thrashing extremo. La cantidad de eventos de fallo de página generados crece extramadamente rápido, ya que cada proceso desaloja las páginas de los demás en un ciclo continuo. En las pruebas

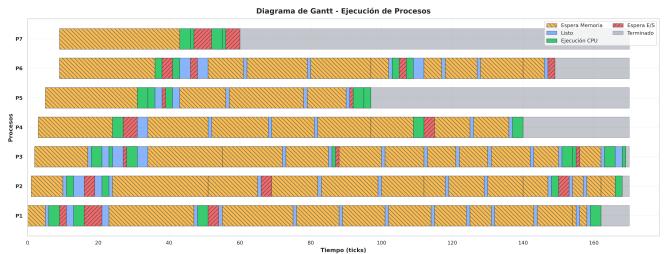


Figura 10. Diagrama de Gantt del caso de estrés de memoria mostrando el impacto de los frecuentes fallos de página.

realizadas, esta configuración generó tal volumen de eventos que la memoria del sistema ejecutándolo (8GB) se saturó antes de completar la ejecución, demostrando cómo el thrashing puede llevar a un sistema a la inoperabilidad total.

V-E. Análisis comparativo

Los cuatro casos demuestran diferentes aspectos del comportamiento del simulador. El caso base establece un punto de referencia funcional donde los algoritmos operan correctamente. El caso de Round Robin ineficiente ilustra la importancia de configurar el quantum apropiadamente según la carga de trabajo. El caso de FCFS expone las limitaciones inherentes de este algoritmo frente a cargas mixtas. El caso de estrés de memoria muestra el impacto de la presión de memoria en el rendimiento y sirve como advertencia sobre los límites operativos del sistema.

VI. CONCLUSIONES

El desarrollo de este simulador permitió integrar los conceptos de planificación de CPU, administración de memoria virtual y gestión de E/S en un sistema funcional. La arquitectura modular del simulador facilita la modificación de algoritmos sin afectar otros componentes del sistema. Esta característica resulta útil para comparar el desempeño de diferentes estrategias de planificación y reemplazo bajo las mismas condiciones de carga.

La implementación de procesos como hilos independientes introduce la complejidad de la sincronización concurrente, pero replica de manera más fiel el comportamiento de un sistema operativo real. El uso de mutex y variables de condición garantiza la consistencia del estado global mientras permite que múltiples procesos existan simultáneamente en memoria. El mecanismo de protección mediante el bit de referencia demostró ser efectivo para prevenir situaciones de inanición cuando un proceso espera la carga de sus páginas en memoria.

Los resultados experimentales validaron el correcto funcionamiento del simulador bajo diferentes escenarios de carga. El caso de escasez de memoria demostró que los algoritmos de reemplazo operan correctamente cuando los recursos son limitados. El caso de alta concurrencia confirmó la estabilidad del sistema con múltiples procesos activos simultáneamente.

El módulo de visualización complementa al simulador proporcionando representaciones gráficas que facilitan la comprensión del comportamiento del sistema. Los diagramas de

Gantt y las gráficas de evolución de colas permiten identificar patrones de ejecución y posibles cuellos de botella. El formato JSONL para la exportación de métricas permite el procesamiento flexible de los datos con herramientas externas.

Como trabajo futuro, se propone extender el simulador para soportar múltiples dispositivos de E/S con diferentes latencias, implementar algoritmos de planificación multinivel con retroalimentación, y añadir la simulación de accesos individuales a páginas de memoria. Esta última extensión permitiría implementar el algoritmo óptimo de manera precisa, evaluando los accesos futuros a páginas específicas en lugar de utilizar la heurística actual basada en el estado de los procesos. También sería interesante agregar una interfaz gráfica interactiva que permita visualizar la simulación en tiempo real.

APÉNDICE

Esta sección presenta los archivos de configuración y definición de procesos utilizados para generar los resultados experimentales del informe.

A. Caso 1: Carga base

```
total_memory_frames=32
frame_size=4096
scheduling_algorithm=RoundRobin
page_replacement_algorithm=LRU
io_scheduling_algorithm=FCFS
quantum=4
io_quantum=4
```

Listing 3. Configuración del Caso 1.

```
P1 0 CPU(5),E/S(3),CPU(4),E/S(2),CPU(4) 1 6
P2 1 CPU(4),E/S(5) 2 5
P3 2 CPU(8) 3 4
P4 3 CPU(3),E/S(2),CPU(3),E/S(2),CPU(3),E/S(1),CPU
    (3) 1 8
P5 4 CPU(2),E/S(1),CPU(2) 2 3
P6 6 CPU(6),E/S(4),CPU(6) 3 7
P7 8 CPU(10) 1 5
P8 9 CPU(4),E/S(2),CPU(4),E/S(2) 2 6
P9 10 CPU(3),E/S(3),CPU(3) 3 5
P10 12 CPU(5),E/S(2),CPU(5),E/S(2),CPU(5) 1 8
```

Listing 4. Procesos del Caso 1.

```
total_memory_frames=40
frame_size=4096
scheduling_algorithm=RoundRobin
page_replacement_algorithm=FIFO
io_scheduling_algorithm=FCFS
quantum=2
io_quantum=2
```

Listing 5. Configuración del Caso 2.

```
P1 0 CPU(20) 1 4
P2 0 CPU(18),E/S(2),CPU(10) 2 5
P3 1 CPU(25) 3 4
P4 1 CPU(15),E/S(1),CPU(15) 1 5
P5 2 CPU(30) 2 6
P6 2 CPU(12),E/S(5),CPU(12) 3 4
P7 3 CPU(20) 1 5
P8 3 CPU(16),E/S(2),CPU(8) 2 4
P9 4 CPU(22) 3 5
```

```
P10 4 CPU(14),E/S(1),CPU(14) 1 4
```

Listing 6. Procesos del Caso 2.

C. Caso 3: Inanición con FCFS

```
total_memory_frames=32
frame_size=4096
scheduling_algorithm=FCFS
page_replacement_algorithm=LRU
io_scheduling_algorithm=FCFS
quantum=5
io_quantum=5
```

Listing 7. Configuración del Caso 3.

```
P1 0 CPU(60),E/S(5),CPU(40) 1 12
P2 1 CPU(2),E/S(1),CPU(2) 2 4
P3 2 CPU(3) 3 3
P4 3 CPU(1),E/S(1),CPU(1) 1 2
P5 4 CPU(4) 2 4
P6 5 CPU(2),E/S(2),CPU(2) 3 3
P7 6 CPU(30),E/S(5),CPU(20) 1 10
```

Listing 8. Procesos del Caso 3.

D. Caso 4: Estrés de memoria

```
total_memory_frames=18
frame_size=4096
scheduling_algorithm=RoundRobin
page_replacement_algorithm=LRU
io_scheduling_algorithm=FCFS
quantum=3
io_quantum=3
```

Listing 9. Configuración del Caso 4.

```
P1 0 CPU(3),E/S(2),CPU(3),E/S(2),CPU(3),E/S(2),CPU
    (3) 1 6
P2 1 CPU(2),E/S(3),CPU(2),E/S(3),CPU(2),E/S(3),CPU
    (2) 2 5
P3 2 CPU(4),E/S(1),CPU(4),E/S(1),CPU(4),E/S(1),CPU
    (4) 3 7
P4 3 CPU(3),E/S(3),CPU(3),E/S(3),CPU(3) 1 6
P5 5 CPU(5),E/S(1),CPU(2),E/S(1),CPU(5) 2 5
P6 7 CPU(2),E/S(2),CPU(2),E/S(2),CPU(2),E/S(2),CPU
    (2),E/S(2) 3 6
P7 9 CPU(4),E/S(4),CPU(4),E/S(4) 1 7
```

Listing 10. Procesos del Caso 4.

REFERENCIAS

- [1] J. L. Peterson and A. Silberschatz, *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1985.
- [2] W. Stallings *et al.*, *Sistemas operativos*. Prentice Hall, 2004, vol. 2.
- [3] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Pearson Education, Inc., 2015.
- [4] cppreference.com, “C++ reference,” <https://en.cppreference.com/>, accessed: 2025-12-02.
- [5] Python Software Foundation, “Python,” <https://www.python.org/>, accessed: 2025-12-02.
- [6] Kitware, “CMake: A powerful software build system,” <https://cmake.org/>, accessed: 2025-12-02.
- [7] Ninja Build, “Ninja, a small build system with a focus on speed,” <https://ninja-build.org/>, accessed: 2025-12-02.
- [8] Catch2, “Catch2: Simple, modern C++ unit testing,” <https://catch2.org/>, accessed: 2025-12-02.
- [9] Just, “Just: A command runner,” <https://just.systems/>, accessed: 2025-12-02.

- [10] Seaborn, “Seaborn: Statistical data visualization,” <https://seaborn.pydata.org/>, accessed: 2025-12-02.
- [11] Gantt.com, “Gantt chart software, information, and history,” <https://www.gantt.com/>, accessed: 2025-12-02.
- [12] Doxygen, “Doxygen: Code documentation. automated.” <https://www.doxygen.nl/>, accessed: 2025-12-02.
- [13] LaTeX Project, “LaTeX: A document preparation system,” <https://www.latex-project.org/>, accessed: 2025-12-02.
- [14] PlantUML, “PlantUML at a glance,” <https://plantuml.com/>, accessed: 2025-12-02.
- [15] clang-uml, “clang-uml: UML diagram generator for C++,” <https://clang-uml.github.io/>, accessed: 2025-12-02.
- [16] RJL.tech, “The definitive JSON lines format resource,” <https://jsonl.rest/>, accessed: 2025-12-02.