UMAG Universidad de Magallanes | Facultad de Ingeniería

Simulador de Gestión de memoria y planificación de procesos: forKing

Milton Hernandez[†], **Iván Mansilla**[†] **y Ayrton Morrison**[†]

[†] *Universidad de Magallanes*

Este informe fue compilado el 25 de octubre de 2024

Resumen

A la hora de trabajar con sistemas operativos es fundamental conocer las Estructuras de Datos que se encuentran involucradas en la gestión de procesos así como los diferentes algoritmos de manejo de estos; materias las cuales facilitan el trabajo del sistema operativo sobre los procesos que los conforman. Por esta razón en el presente informe se describe una simulación simple realizada sobre el lenguaje C de un gestor de memoria y un manejador de procesos con estructuras de datos tales como: Listas enlazadas simples, Colas y Árboles Binarios.

Keywords: *Estructuras de datos, LES, Buddy System, Colas, forKing*

Índice

1. Introduction

Forking es un simulador de gestión de memoria y planificación de procesos, este hace uso de diversas estructuras de datos, como lo son; Listas en enlazadas simples, colas, arboles, etc. El simulador se encarga de emular la ejecución de procesos en un sistema operativo, pasando por una asignación de memoria, para luego ser procesado con criterios específicos para maximizar la eficiencia del sistema.

Para llevar a buen término el desarrollo de esta aplicación hubo que decidir cuales serían las características que harían de este proyecto uno de interés. Algunas a destacar son:

1. **Cantidad de procesos**: Es importante permitir al usuario especificar el numero de tareas que se encuentran en el sistema, logrando procesar cantidades pequeñas pero también generar simulaciones con cantidades mayores.
2. **Tiempo de llegada y tiempo de ejecución**: Es fundamental darle libertad al usuario para especificar el tiempo de llegada de cada proceso, así como el tiempo de ejecución de cada tarea, para hacer simulaciones fieles al comportamiento real.
3. **Memoria requerida por cada proceso**: Por supuesto las tareas no ocuparán todas la misma cantidad de memoria RAM por lo que manejar memorias diversas será importante para el buen desarrollo del proyecto.
4. **Fragmentación de memoria**: Controlar la fragmentación de memoria puede ser clave para la eficiencia de un sistema operativo, por lo que fue una de las prioridades a la hora de desarrollar esta aplicación.

La elección de los algoritmos de gestión y planificación de procesos fue un punto clave en el desarrollo del proyecto, ya que estos conforman las bases del simulador y son los encargados de hacer que el sistema sea eficiente y funcional. Los algoritmos de gestión de memoria y planificación de tareas elegidos fueron:

1. **BuddySystem**: Algoritmo de gestión de memoria que divide la memoria en bloques de tamaño potencia de 2, permitiendo una asignación de memoria eficiente.
2. **RoundRobin**: Algoritmo de planificación de procesos que asigna un tiempo de ejecución a cada proceso, permitiendo que todos los procesos tengan la misma prioridad.
3. **ShortestJobFirst**: Algoritmo de planificación de procesos que trabaja proceso que requieren de un *burstime* menor, permitiendo que los procesos mas cortos tengan prioridad.

Mas adelante se profundizará más en el funcionamiento e implementación de cada uno de estos, asi como también en las estructuras de datos utilizadas.

2. Objetivos

El objetivo principal de este proyecto es: Construir una simulación de sistema de gestión de memoria y planificación de procesos, mediante el uso de las estructuras de datos vistas en clase.

2.1. Objetivos secundarios

1. **Uso de estructuras de datos**: Construir un software funcional, eficiente y bien estructurado, mediante el uso de estructuras de datos, como listas enlazadas, colas, arboles, etc.
2. **Eficiencia**: Generar breves tiempos de procesamiento haciendo un uso eficiente de recursos y una elección adecuada de algoritmos.
3. **Trabajo en grupo**: Reforzar el trabajo en equipo, asignando roles y tareas a cada miembro del equipo.
4. **Planificación de tareas**: Dividir el proyecto en tareas atómicas que permitan su realización de manera más efectiva, asignando tiempos y recursos a cada una de estas.

3. Planteamiento del desarrollo del proyecto

Para llevar a cabo el proyecto forKing fue necesario comprender cuál era el objetivo final al que se deseaba llegar razón por la cuál se ha decidido dividir el trabajo en dos fases: Gestión de memoria y planificación de procesos.

3.1. Gestión de memoria

En este punto fue necesario plantearse la manera en que se simularía la memoria dentro de forKing; ¿Se usaría memoria real a través de `Malloc()`? o tal vez ¿Podemos simular la memoria con un *arreglo estático* donde asignar a cada uno de sus elementos una "cantidad de memoria" determinada?

Luego de investigar la gestión de memoria en diversos sistemas operativos se optó por hacer una implementación sencilla del sistema usado por `Linux`, el **BuddySystem** el cuál consiste en lo siguiente:

- Se asume la memoria como un gran bloque de espacio disponible, supondremos un ejemplo con 2048 bytes de memoria.
- Esta memoria construida tiene la capacidad de dividirse de manera *binaria* recursivamente, lo que quiere decir, en el caso del ejemplo, que se pueden tener:
  - Hasta 1 bloque de 2048 bytes
  - Hasta 2 bloques de 1024 bytes
  - Hasta 4 bloques de 512 bytes
  - Hasta 8 bloques de 256 bytes
  - Hasta 16 bloques de 128 bytes
  - Hasta 32 bloques de 64 bytes
  - Hasta 64 bloques de 32 bytes
  - Etc...
- Cuando un proceso requiere entrar en el BuddySystem se busca el bloque de memoria más cercano a la cantidad de memoria requerida (por encima) y asigna la memoria a dicho proceso, por ejemplo, si llegara un proceso que requiere de 500 bytes de memoria, se asignaría un bloque de memoria de 512 bytes, ya que este es el tamaño de bloque más cercano.
- Es aquí donde la división entra en juego. Si no existe un bloque de 512 bytes de memoria, pero sí uno de 1024 bytes, entonces este se dividirá en dos bloques de 512 bytes, uno de los cuales será ocupado por el proceso, y el otro quedará libre para ser usado o fragmentado de ser necesario.
- Cuando un bloque se divide en dos más pequeños diremos que estos son *Buddys* el uno del otro.
- En caso de que un proceso termine de usar un bloque de la memoria este queda libre. En este momento se evalúa si su *Buddy* también está libre y, de ser así ambos se juntan generando un bloque más grande.

> **Nota**
> Cada bloque dentro del *BuddySystem* cuanta con una característica conocida como **Orden del bloque**, donde:
>
> Un bloque que no se puede dividir tendrá orden 0 Y el orden irá ascendiendo hasta llegar al bloque más grande posible que tendrá el orden más alto posible.

¿Cuales son las ventajas de la utilización de un *BuddySystem*? Principalmente que este sistema permite manejar de manera muy eficiente el fenómeno de la **Fragmentación de memoria** que ocurre que cuando procesos que requieren poca memoria terminan de ser procesados dejando partes muy pequeñas de la memoria vacías, lo que genera que, cuando un proceso que requiera de mucha memoria desee ser procesado, no encuentre un lugar lo suficientemente grande para ubicarse dentro de la memoria.

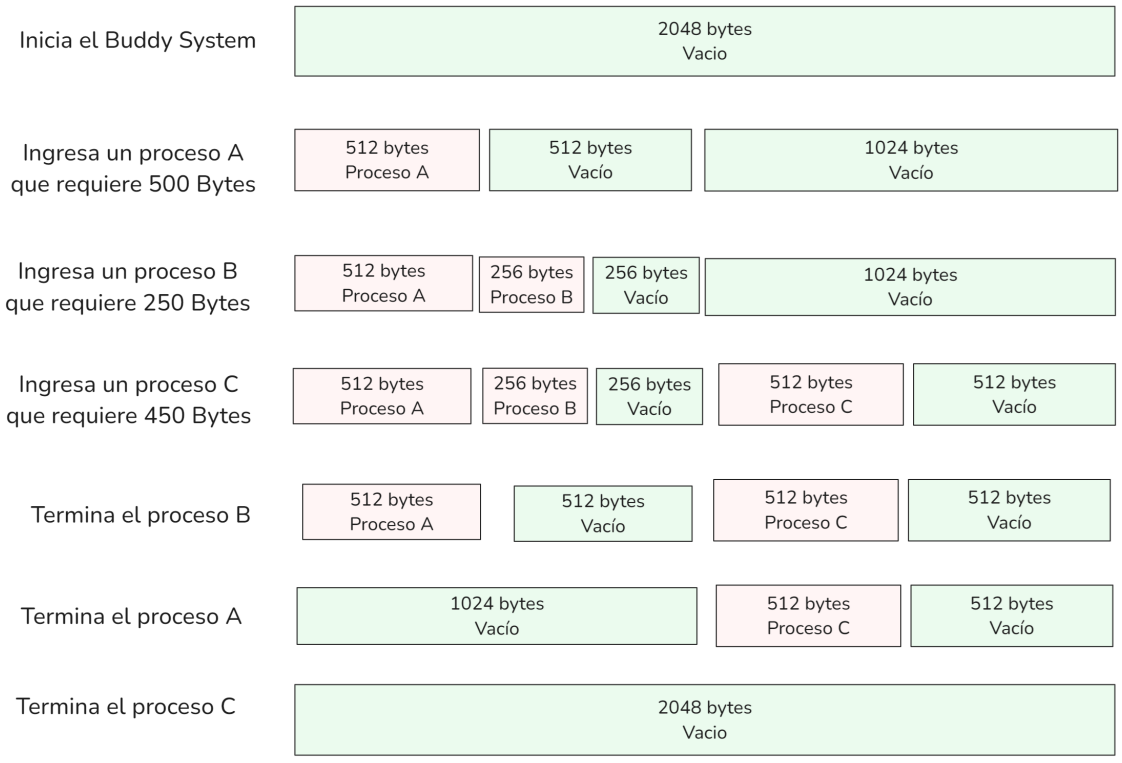

Figura 1. Ejemplo de funcionamiento del buddySystem

En la figura1 se puede ver el comportamiento de un *BuddySystem* de este tipo de manera gráfica.

3.2. Manejo de procesos

Una vez que ya tenemos una lista de procesos listos para su ejecución necesitamos de una manera eficiente para procesarlos. Una posible opción sería usar el algoritmo `FCFS` que consiste en atender a los procesos en el orden de llegada, sin embargo no es del todo eficiente, puesto que si llega un proceso que tarda 100*s* en ser ejecutado y detrás viene uno que tarda 10*s* este segundo deberá esperar demasiado tiempo para su ejecución.

Por esta razón surge el algoritmo `Shortest Job First` que consiste en atender a los procesos en el orden de su tiempo de ejecución (`burstTime`), sin embargo esto puede provocar que los procesos con mucho tiempo de ejecución no sean ejecutados nunca.

Para solucionar este último inconveniente aparece el algoritmo `Round Robin` donde todos los procesos sean parcialmente atendidos, dando como mínimo a cada uno *n* ráfagas de CPU (`ticks`), donde *n* es conocido como `quantum`. Así no existen procesos con tiempos de ejecución muy altos que no sean ejecutados.

En el presente proyecto se optó por el uso combinado de los algoritmos `Shortest Job First` y `Round Robin` en un modelo conocido como **algoritmo de colas multinivel** donde para un mismo Sistema Operativo se usan diversas colas que permiten un procesamiento eficiente de los recursos. El funcionamiento que se decidió dar al programa es el siguiente:

- Un proceso cuyo `burstTime` es menor al `quantum` designado entra directamente a la cola `sjfQueue`.
- Un proceso cuyo `burstTime` es mayor o igual al `quantum` va a una cola que recibe el nombre de `rrQueue`.
- Si el `quantum` termina y el proceso sobre el que se trabajó tiene un tiempo restante menor al `quantum` va directamente a la `sjfQueue`.

- Por último la `sjfQueue` tiene más prioridad que la `rrQueue`, de forma que los procesos cortos se atienden primero y los largos se atienden mediante el `Round Robin`.

3.3. Planificación general del proyecto

Conociendo ya los conceptos usados en el proyecto resta solamente comentar cómo será la ejecución del programa, sin embargo antes se aclararán algunos conceptos adicionales propios del proyecto.

- **ArrivalQueue**: Es una cola donde se almacenan los procesos que llegan a la simulación (Se asume que un proceso llega a la simulación si su `arrivalTime` coincide con la cantidad de `ticks` que han transcurrido desde que se inició el programa).
- **WaitingQueue**: Es una cola donde se almacenan los procesos que están esperando a ser procesados (Un proceso entra a esta cola si no existe memoria en el `BuddySystem` para este).
- **RRQueue**: Es una cola donde se almacenan los procesos que deben ser procesados por `Round Robin`.
- **SJFQueue**: Es una cola donde se almacenan los procesos que deben ser procesados por `Shortest Job First`.
- **BuddySystem**: Estructura de datos que representa el *sistema* de buddies descrito anteriormente.

Con todas estas herramientas se describe el funcionamiento de *forKing* en la figura 2 donde se omiten comprobaciones varias de las colas para simplificar el diagrama.

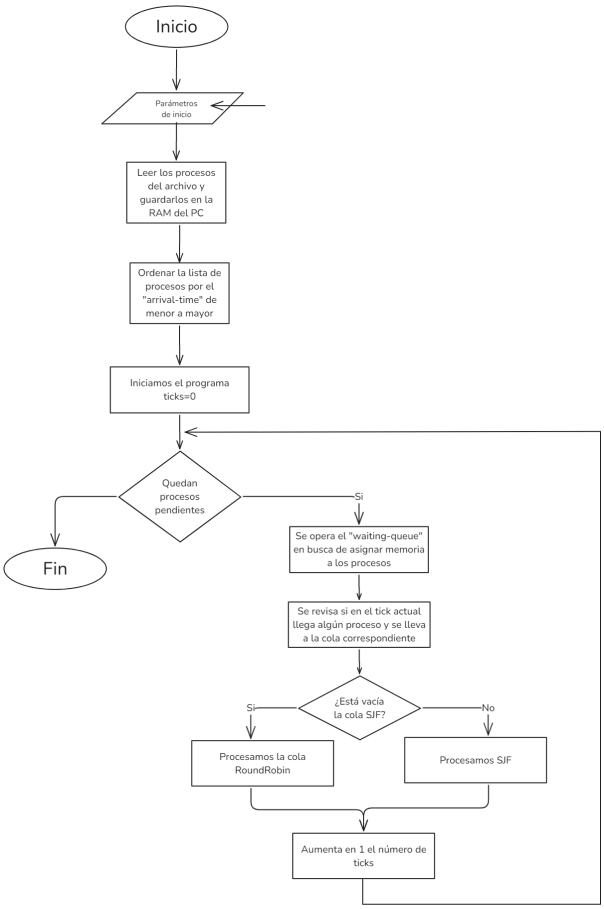

Figura 2. Diagrama de flujo del desarrollo de forKing

4. Implementación

La implementación de los diversos algoritmos y estructuras de datos fue realizada en el lenguaje de programación C. Las diferentes funciones y estructuras fueron implementadas en archivos separados, con el objetivo de mantener una estructura de directorios clara y ordenada. A continuación se detallara esta implementación.

4.1. Estructura de directorios

Dentro del directorio `src` se encuentran los siguientes archivos:

- `BuddySystem.c`: Contiene la implementación del algoritmo de asignación de memoria Buddy System.
- `errors.c`: Contiene la implementación de funciones para manejar errores.
- `files.c`: Contiene la implementación de funciones para manejar archivos.
- `forking.c`: Contiene las funciones para manejar `main` dentro del programa, utilizando las demás funciones y estructuras del proyecto.
- `main.c`: Contiene la función `main` del programa.
- `process.c`: Contiene la implementación de funciones y estructuras de datos para manejar procesos.
- `queue.c`: Contiene la implementación de funciones y estructuras de datos para manejar colas.
- `utilities.c`: Contiene la implementación de funciones varias, de utilidad.

Dentro del directorio `testing`, se encuentran los archivos:

- `generator.c`: Contiene la función encargada del generador de "procesos".
- `gant_creator.py`: Es un script de **Python** encargado de generar una carta gantt.

Por ultimo, dentro del directorio `incs` se encuentran las cabeceras incluidas en todos los archivos `.c`.

4.2. Estructuras de datos

Las estructuras de datos utilizadas fueron las siguientes:

Lista enlazada simple

Esta se encuentra implementada con la estructura `ProcessNode`, el cual contiene las variables:

- `Process data`: Donde se almacenarán los procesos
- `Position next`: Puntero al nodo siguiente de la lista

La lista enlazada simple se utilizo para almacenar y ordenar los procesos ingresados en el programa según su *arrival time*, el ordenamiento de la lista se realizo en primer instancia con *bubble sort*, sin embargo al probar el programa en cantidades grandes, lo ineficiente de este algoritmo salio a flote, debido a que se encontró con tiempos de espera extensos al ejecutar el programa. A raíz de de esto se decidió implementar el algoritmo *merge sort*, el cual redujo sustancialmente la cantidad de tiempo de demora del programa en ordenar los datos.

Esta estructura de datos es la encargada de almacenar los procesos en memoria, con esto las demás estructuras unicamente deben utilizar punteros a estos datos, ahorrando una cantidad significativa de espacio.

Colas

Las colas se encuentran implementadas con la estructura `CircularNode`, la cual, en realidad es una *lista circular doblemente enlazada* en donde el *rear* y el *front* de la cola son representadas por el nodo anterior y siguiente del centinela de la lista. Este contiene las variables:

- `Process* process`: Puntero al proceso almacenado. Este puntero apunta a algún proceso contenido en la lista enlazada simple anterior
- `PtrToCircularNode next`: Puntero al nodo siguiente de la lista
- `PtrToCircularNode prev`: Puntero al nodo anterior de la lista

La implementación de las colas fue fundamental para el funcionamiento del programa, ya que sus características permiten manejar fácilmente el flujo y orden de los procesos. Las colas fueron usadas en el:

- Arrival Queue
- Waiting Queue
- SJF
- RoundRobin

Para insertar elementos en las colas existe la función estándar de `enqueue`, la cual simplemente ingresa un elemento en la cola, sin embargo por motivos de comodidad y para cumplir con el funcionamiento deseado se incluyo la función `sorted_enqueue`, la cual se encargo de ingresar decreciente o crecientemente un proceso en cola según un criterio dado. Por ejemplo al ingresar en SJF se ingresa según su *burstime*.

Arboles binarios

Los arboles binarios se encuentran implementados en la estructura `_treeNode` , definida por las siguientes variables:

- `Buddy element`: Corresponde a una estructura la cual contiene un puntero a proceso, la altura del elemento dentro del arbol y una banera que indica el estado de uso del elemento, representadas por: `Process* process`, `unsigned int order` y `isUsed` respectivamente.
- `TreePosition parent`: Puntero a nodo el cual representa al nodo padre.
- `TreePosition left`: Puntero a nodo que representa al hijo izquierdo.
- `Treeposition right`: Puntero a nodo que representa al hijo derecho.

El árbol binario fue implementado en el *BuddySystem*, asignando "memoria" inteligentemente mediante el operaciones relacionadas a las potencias de dos y logaritmos en base dos, con el fin de decidir el espacio optimo del proceso entregado según su memoria requerida. Inicialmente estas operaciones se realizaron matemáticamente, mediante funciones de la biblioteca `math.h` sin embargo se opto por realizarlas de forma binaria, lo que disminuyo la carga del programa, haciéndolo mas eficiente.

4.3. Algoritmos de planificacion

Como ya fue mencionado anteriormente los algoritmos de planificación utilizados fueron `Shortest Job First y Round Robin`. Su implementación consistió en:

- **Shortest Job First**: Para la implementación del algoritmo *SJF* se utilizo una cola a través de una lista circular doblemente enlazada, esta cola se ordeno mediante la función `sorted_enqueue`, realizando comparaciones con el *burstime* del proceso que esta siendo trabajado (en caso de que la cola que este siendo procesada sea *SJF*) con el proceso ingresado.
- **Round Robin**: En el caso de la cola *Round Robin*, esta se trabajó con la misma estructura de datos que el *SJF*, sin embargo el funcionamiento de esta consiste en revisar el *burstime* del proceso a ser trabajado, se realiza una comparación para ver si es igual o mayor que el *Quantum*, de

ser asi se asigna a esta cola y este proceso es trabajado la cantidad de ticks que indica el *Quantum*. Al llegar al final del *Quantum* se verifica si el proceso debe pasar o no a la cola *SJF* mediante una comparación simple.

4.4. Otros ejecutables

Otros programas utilizados en la ejecución del programa fueron un generador de procesos y parámetros y un generador de carta gant. El primer programa corresponde a un generador simple de procesos, el cual entrega parámetros aleatorios entre ciertos valores dados por el usuario mediante terminal.

```
5000 2 4 5

1 20 3849
4 1 867
10 3 3271
12 3 4859
10 4 187
14 6 4859
8 2 83
11 2 1577
```

Código 1. Ejemplo de entrada forKing

Mientras que el generador de carta Gantt utiliza el lenguaje de programación **Python** con las librerias `matplotlib, numpy y random`, leyendo un archivo tipo csv el cual se genera al ejecutar el programa principal.

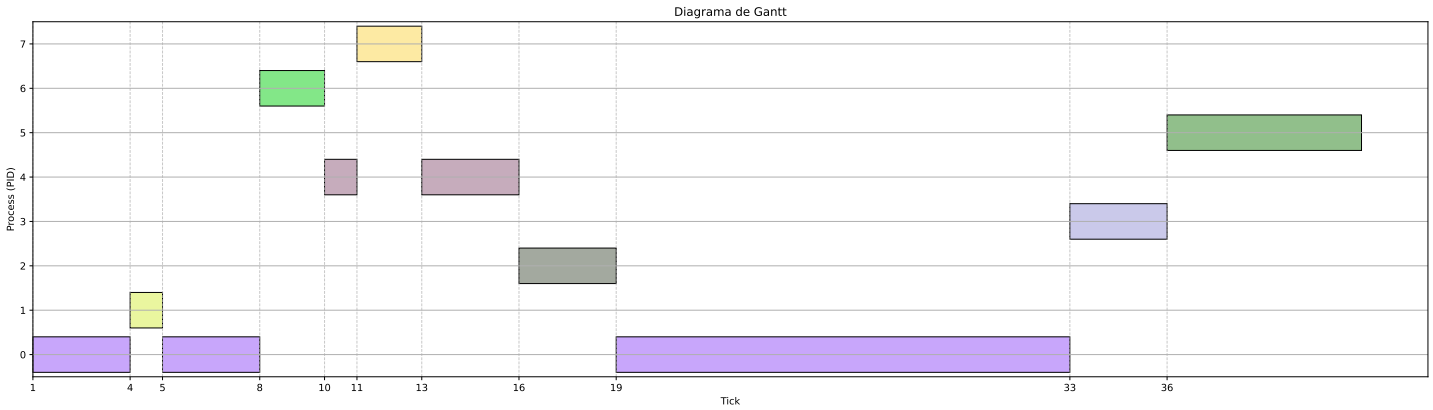

Figura 3. Ejemplo de carta Gantt con forKing

5. Gestión del equipo de trabajo

El equipo de trabajo constó de 3 personas, entre las cuales se tomaron las decisiones y se distribuyeron las tareas. Para un mejor orden y eficiencia, se establecieron normas de codificación, se acordó trabajar mediante una lista de tareas pendientes y realizar reuniones semanales.

5.1. Normas de codificación

En lo que respecta el nombramiento de variables, las constantes fueron llamadas con el formato `SCREAMING_SNAKE_CASE`, y las variables con `camelCase`.

```
#define CLEAR_SCREEN "\033[H\033[J"
unsigned int arrivalTime;
```

Código 2. Ejemplo de Screaming Snake Case y camelCase

Se acordó nombrar a las funciones con el formato `snake_case`, y colocar sus llaves de apertura debajo de su prototipo.

```
void delete_list(List L)
{
    Position P, Tmp;
    P = L->next;
    L->next = NULL;
    while(P!= NULL){
        Tmp = P->next;
        free(P);
        P = Tmp;
    }
}
```

Código 3. Ejemplo de Snake case

Por otro lado las llaves de apertira de otros bloques de código se colocaron justo al lado de su línea final, evitando omitir las llaves cuando existiese una única sentencia.

```
if(!clean){
    // Imprimir Informacion final
    print_program(&forkingConfig, forkingStatus);
}
```

Código 4. Ejemplo bloques de código

Por último, para los archivos de cabecera se acordó usar guardias, con el nombre de su archivo en `SCREAMING_SNAKE_CASE`.

```
#ifndef PROCESS_H
#define PROCESS_H
```

Código 5. Ejemplo de Guardias en archivos de cabecera

5.2. Lista de tareas y organización

Una vez establecida una norma de codificación común, se establecieron objetivos a corto, mediano y largo plazo, los cuales fueron listados para mantener un control del flujo de trabajo y repartirlos; por ejemplo, el primer objetivo fue crear las estructuras de datos necesarias (colas, listas enlazadas y árbol binario), por lo que se le encargó a cada integrante la codificación de una de ellas.

5.3. Reuniones semanales

Se fijaron semanalmente días y horarios de trabajo en conjunto, con el objetivo de discutir y/o fijar los objetivos del proyecto, resolver problemas de código mayores, o simplemente avanzar con Las tareas fijadas cada uno por separado; estas reuniones se hicieron de forma tanto presencial como asincrónicas a través de videollamadas y la herramienta LiveShare[1] para codificar en conjunto. Esta modalidad fue extremadamente útil, pues se mantuvo una buena consistencia y flujo de avances semanales, en poco tiempo cumpliendo en poco tiempo con buena parte de los objetivos autoimpuestos. Por otro lado, para tareas menores se permitió que cada uno trabajara individualmente en la tarea que estimase conveniente fuera de estos horarios establecidos.

[1] Extensión de Visual Studio Code que permite el trabajo paralelo en un entorno virtual.

6. Posibles Mejoras a futuro

A pesar de su correcta funcionalidad *forKing* tiene algunas limitaciones que podrían ser mejoradas en el futuro así como caracteristicas que podrían mejorar su funcionalidad pero que, por distintas razones, fueron omitidas; estas son las siguientes:

1. **Ordenamiento de colas**: La cola `sjfQueue`, para su correcto funcionamiento debe estar ordenada de tal manera que el proceso que se encuentra al inicio de la misma sea aquel de la cola con el menor `burstTime`; este es también el caso de la cola `WaitingQueue` donde el primer proceso debe ser el de menor requerimiento de memoria. El proceso usado para conseguir este resultado tiene una complejidad de $O(n)$, sin embargo no se encontró una manera más eficiente de conseguirlo. Lo anterior ocasiona que para cantidades de procesos muy grandes la simulación pueda tardar mucho tiempo en ejecutarse.
2. **Uso ineficiente del BuddySystem**: El *BuddySystem* a pesar de solucionar la mayoría de problemas de asignación de memoria tiene un defecto clave: Depende de espacios con potencias de dos, esto ocasiona que si un proceso requiere 1025 bytes de memoria, el *BuddySystem* tendrá que asignar 2048 bytes de memoria a dicho proceso, desperdiciando 1023 bytes de memoria. Esto podría solucionarse complejizando más el algoritmo de asignación de memoria.
3. **Trabajo con hebras y/o procesos paralelos**: Una de las ideas más ambiciosas surgida en los inicios del desarrollo de forKing fue el trabajar con procesos paralelos, mediante el uso de *hebras*[2] o, de procesos generados con `fork()`[3] (de ahí el nombre del proyecto), sin embargo dada la complejidad de esta implementación y el tiempo con el que se contaba para la realización del proyecto se prefirió no enforcar los esfuerzos en dicha dirección.
4. **Bloqueo de procesos, espera y ráfagas de E/S**: Durante el inicio de este proyecto se realizó una simplificación de lo que un simulador de un gestor de memoria y planificación de procesos debería hacer, en esta etapa se simplificó su funcionamiento dejando de lado algunas características como:
    - **Bloqueo de procesos**: Se decidió no tener en cuenta el bloqueo de procesos, donde un proceso podría necesitar de la previa ejecución de otro para proceder con su ejecución.
    - **Ráfagas de E/S**: En sistemas operativos reales un proceso no necesita un único *burstTime* para su completa ejecución sino que puede requerir de varias ráfagas de CPU combinadas con ráfagas de E/S, por lo que se decidió no tener en cuenta esta característica.

7. Conclusiones

A raíz de la realización de este proyecto se pudo comprender de mejor manera los usos reales de las estructuras de datos analizadas durante las clases. Además fue posible evidenciar la importancia de los conceptos introducidos por estas estructuras para el desarrollo de nuevas estructuras como el caso del *BuddySystem*.

Puesto que este proyecto permitió el trabajo con grandes volúmenes de datos se comprendió de mejor manera los conceptos de eficiencia de algoritmos como el `Merge Sort` en contra de los algoritmos de ordenamiento de listas como `Bubble Sort`. Así como la importancia de la creación de funciones eficientes para el manejo de las diferentes estructuras de datos.

Además, al tratarse de una simulación de un gestor de procesos y de memoria se pudo comprender mejor el funcionamiento de algoritmos como `Round Robin` y `Shortest Job First`, los cuales son usados en muchos de los sistemas operativos modernos.

[2] Unidad más pequeña de ejecución dentro de un proceso, que permite realizar tareas concurrentes compartiendo la memoria del proceso principal
[3] Funcion estandar de la libreria “unistd.h”, la cual permite generar procesos hijos. [2]

Referencias

[1] A. S. Tanenbaum y A. S. Woodhull, *Sistemas Operativos: Diseño e implementación*, Segunda edición, trad. por R. Escalona. México: Prentice Hall, 1997.

[2] T. O. Group. «fork() - create a new process». (2004), dirección: https://pubs.opengroup.org/onlinepubs/009696799/functions/fork.html (visitado 12-10-2024).

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, *Introduction to Algorithms*, 3rd. Cambridge, MA: MIT Press, 2009, ISBN: 978-0-262-03384-8.

[4] geeksforgeeks. «Buddy System – Memory Allocation Technique». (sep. de 2024), dirección: https://www.geeksforgeeks.org/buddy-system-memory-allocation-technique/.